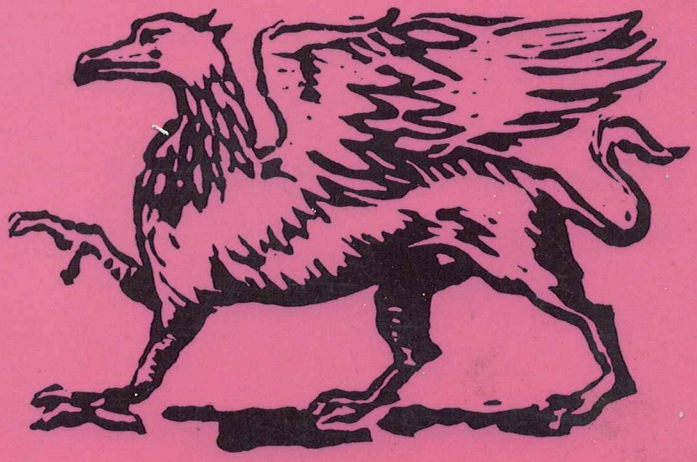
Dr. Dobb's Journal

For Users of Small Computer Systems

A Reference Journal for Users of Home Computers

Volume Seven



The People's Computer Company Series

DR. DOBB'S JOURNAL

For Users of Small Computer Systems

A Reference Journal for Users of Home Computers

Volume Seven

The People's Computer Company Series

HAYDEN BOOK COMPANY
a division of Hayden Publishing Company, Inc.
Hasbrouck Heights, New Jersey

Acknowledgments

As always, the allegiance of all editors at *Dr. Dobb's Journal* goes to the many people whose support has kept the pages of *DDJ* well fed. Our naive, optimistic beginnings were in the days when the microcomputer was still a somewhat dubious creation in the eyes of the mainframe elite, and was a virtual unknown to the public at large. Since then, it has become fashionable to clamber aboard one of the several popular bandwagons filled to overflowing with look alikes and cheaper imitations of soon-to-be-mediocre products.

So we offer wholehearted thanks to all the authors and readers who have made *DDJ*'s seventh year of publication as exciting and important as its first. You have shown us that there is, more than ever, a tremendous need for blue-sky thinking, for a champion of the public domain, for a publication whose philosophy addresses human needs and the potential computers have to serve them. Your enthusiasm has been our inspiration.

Marlin Ouverson
Bound Volume Editor
March 1, 1984

Bound Volume Staff
Editor: MARLIN OUVERSON
Production Manager: JANE A. McKEAN
Index: JULIE ANTON

Cover: Linocut of griffin by Wendy Palmer. The mythological griffin, half lion and half eagle, has long stood as an enemy of ignorance.

Copyright © 1984 by People's Computer Company. All rights reserved. No part of this book may be reprinted, or reproduced, or utilized in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying and recording, or in any information storage and retrieval system, without permission in writing from the Publisher.

Printed in the United States of America

2	3	4	5	6	7	8	9	PRINTING
85	86	87	88	89	90	91	92	YEAR

Editor's Preface

This is the "Blue Period" volume.

Even the most casual observer of the microcomputer scene in 1982 knew the year belonged to IBM. It began with the Justice Department dropping a longstanding antitrust suit against "Big Blue" and ended with the computer newsweekly *InfoWorld* selecting IBM's first personal computer as Product of the Year.

IBM had introduced the PC the preceding August and by January 1982 the entire microcomputer software industry was already in IBM's long, blue shadow. By March the PC-compatible hardware products were multiplying. Before Christmas, IBM had driven a wedge between Microsoft and Digital Research, caused a half-year delay in the release of a new version of what had been the industry-standard operating system, and made Apple Computer, Inc. revamp its entire marketing strategy. In short, the billion-dollar multinational corporation, with firm notions of corporate secrecy and a reputation for letting other companies take innovative risks, made very big waves that year. Those waves altered an industry that had been created from nothing over the preceding decade by a bunch of hobbyists and wild-eyed dreamers accustomed to openly sharing technological ideas.

Among the swimmers watching the Big Blue wave roll in that year was *Dr. Dobb's Journal,* a magazine founded back in the early months of the microcomputer revolution. *DDJ* continues the open sharing of updates on software tools as well as the promotion of clever, tight, innovative programming. The founders of the journal believed that the personal computer could and should be an unparalleled equalizing force in society, and that putting the best software tools in the hands of the brightest programmers was the quickest path to this goal. They were revolutionaries, while IBM had a somewhat different agenda.

This volume will inform you of the plentiful software tools now available. You'll find Dave Cortesi's comparison of the leading operating systems for the PC—the first and probably still the best analysis of CP/M-86 and MS-DOS available. Then there are Ray Duncan's articles on the 8087 math co-processor, color graphics for the PC, and his 8086 Forth assembler. Ray's 8086 articles were so impressive that he was commissioned to write a regular column to promote the discussion and interchange of 16-bit software. Advanced processors such as the 8086 and 68000 were beginning to fascinate programmers in 1982 and *Dr. Dobb's Journal* increased its attention to them dramatically without abandoning the CP/M world. Bob Blum's popular CP/M Exchange column also began in 1982. In case you're not familiar with *Dr. Dobb's Journal,* I should point out that there are full implementations of languages and operating systems between these pages, as well as other useful and significant utility programs and software.

But this volume is history, too. The material herein was originally written as the first phase of the personal computer revolution came to an end. As we have always done in these volumes, we've given you the full year of *Dr. Dobb's Journal:* editorials, letters to the editor—the works. You'll see attitudes expressed by *DDJ*'s readers to IBM's entry into the market. In fact, the magazine's original subtitle was "Running Light Without Overbyte" (that is, programming so well as not to waste a byte of precious computer memory). One reader characterized IBM's system software as "overbyte galore!" You may also read a blue wistfulness between the lines for the "ending era" in Marlin Ouverson's editorials. And you'll read of his concern for "the concept of cooperation and... the attitude that anything is possible and should be tried at least once." These are principles that, Marlin says, "begat personal computing and are the parents of *DDJ*." That's not rhetoric; it's just how it was.

No, the Big Blue wave didn't wash *DDJ* away, and cooperation and idealism didn't die in 1982. The hobbyist era ended, but the revolution lived on. IBM is now part of it and its effect on the personal computer revolution has been largely salutary. But anything as big as IBM is by definition a monster, and not to be relied upon.

So the Doctor's work goes on: to look for and distribute the best software tools for advanced programmers and to provide a forum for discussing the merits of various programming methods. Personal computers *can* be the great equalizer, but the revolution needs nudging now and then. Any revolution with an IBM in it needs a *Dr. Dobb's Journal.*

Michael Swaine
Editor-in-Chief
Dr. Dobb's Journal
April 1984

About People's Computer Company

People's Computer Company, the publisher of *Dr. Dobb's Journal*, was founded in 1972 as a non-profit source of education and information about computers. It grew from the same roots as the *Whole Earth Catalog* and the *Whole Earth Software Catalog*. Both companies were offshoots of the Portola Institute of Menlo Park, California, an organization investigating and promoting the idea of alternative technology. For the founders of PCC, this meant promoting the use of computers as a liberating force in society.

To make these ideas manifest, it has been necessary to think far ahead, to be creative and innovative. PCC published the first personal computing periodical in 1972, a time when the technical community at large found the thought of "personal" computers preposterous.

PCC has initiated a number of service-oriented projects in order to empower people to take a more active part in a society in which computers are a significant factor. It has published a newspaper, three magazines, and several books on home and personal computing. It has also operated a neighborhood computer center, a mail-order bookstore, a computer van, and has sponsored telecommunications research (PCNET).

The main emphasis of the company has been on small, personal computers—the kind anyone can learn to use and that increasingly more individuals can afford. ComputerTown, USA! is a major project begun with financial assistance from the National Science Foundation. It provides a model of how to make everyone in a small- to medium-size community "computer literate." ComputerTown shares its experience and resources in a newsletter in the ComputerTown Books series, in the ComputerTown Guidebooks, and through classes and membership services. Affiliates exist in the United Kingdom, Europe, Australia, and across the United States.

PCC today continues its consulting leadership in technology research, curriculum design, and training. In conjunction with the California State Library, PCC is working on the design of written materials supporting the creation of 12 public library resource centers in the state of California. The staff of PCC will provide the on-site training for the staffs of these centers and for the staffs of 71 additional public libraries. PCC continues its focus of developing a model of science as part of a continuing dialog with those affected by it.

Dr. Dobb's Journal, which was licensed to M&T Publications of Palo Alto, California in 1984, was PCC's second publication. The first issue was published in 1976 in order to provide short-term distribution of the newly written Tiny BASIC language. Reader response was so strong that it went immediately into monthly production. It has led the way since then by focusing on important advances in microcomputer software, printing public domain software, and fostering vital reader interaction on technical matters. Most readers describe their experience with computers as intermediate to extensive. About ten percent of them live outside the United States.

Today *Dr. Dobb's Journal* remains the prime source of information and software tools for advanced programmers. Its early issues are still in demand, and no part of the editorial content of *DDJ* has ever gone out of print. Volumes One through Seven of this series contain the entire editorial content of *DDJ*'s first seven years, and are available in matching, bound volumes from Hayden Book Company.

The PCC staff members have a wide variety of personal and professional interests and experiences. Their work stems from a conviction that the personal computing revolution is of tremendous significance, and that familiarity with computers fosters confidence and greater control over one's life.

Bob Albrecht, one of the company's founders, explained the inspiration for the company in the first issue of the *PCC Newspaper:*

"Computers are mostly used against people instead of for people, used to control people instead of free them. It's time to change all that. We need a People's Computer Company."

People's Computer Company
P.O. Box E
Menlo Park, CA 94025

Contents

VOLUME 7, NUMBER 63, 1982

2 Editorial
3 Dr. Dobb's Clinic
6 Of Interest
8 Interfacing the 68000 to an AIM 65 — Luca Fusina and Claudio Granuzzo
13 A Portable Screen-Oriented Editor — Edward K. Ream
20 Small Shell: Part 2 of a North Star VOS — J. E. Hendrix

VOLUME 7, NUMBER 64, 1982

54 Editorial
55 Letters
57 Of Interest — D. E. Cortesi
59 Recursion in FORTH — Glyn Emery
61 FORTH 8086 Assembler — Ray Duncan
64 Formal Definition of FORTH as a Programming Language — C. H. Ting
86 Excuse Me, What Time Is It? — Alan Bomberger
91 Background Information on Text Editors — Leonard K. Nicholson
93 Standards for Computerized Measurement — Douglas Crockford
95 Software Review
"North Star Management Program" — Gene Head
96 Book Reviews
"The Devil's DP Dictionary" — Joey Robichaux
"LISP" — Brian Gordon

VOLUME 7, NUMBER 65, 1982

98 Editorial
99 Letters
101 Dr. Dobb's Clinic — D. E. Cortesi
104 Of Interest
106 Runic: An Interactive, Extensible Compiler — Marty Franz
109 Put the H-8 Front Panel to Work — D. C. Shoemaker
111 Improvement Upon a Division Program by Leventhal — W. D. Maurer
114 A Pattern Classification Program — K. F. Wegner and D. E. Raeside
116 Pidgin for the 8080 Under CP/M — Andrew L. Bender
141 Book Reviews
"The 8086 Book (Includes the 8088)" — Ray Duncan
"Pascal Programming Structures" — Richard L. Lozes
"Build Your Own Z80 Computer" — George W. Jolly
143 Software Review
"The WORD" — Gene Head

145 Hardware Review
"Universal Data Systems 103LP Modem" DAVE CAULKINS

VOLUME 7, NUMBER 66, 1982

149 Letters
149 Editorial
152 Dr. Dobb's Clinic D. E. CORTESI
154 Of Interest D. E. CORTESI
156 8080/8086 Cross-Assembler, Part 2 JONATHAN MILLS
165 Small-C DISKDOC: A Repair and Maintenance Utility EGIL KVALEBERG
172 Make the Most of Your Check Digits PETER ZILAHY INGERMAN
175 A Poor Person's Spelling Checker ALAN BOMBERGER
184 Son of Runic: Writing the Compiler MARTY FRANZ
187 Software Reviews
"PIE" D. C. SHOEMAKER
"Video Scribe" D. C. SHOEMAKER
188 Hardware Review
"The Sinclair ZX81" JERRY CHAMKIS
189 Book Reviews
"The Logical Design of Multiple-Microprocessor Systems"
GEORGE W. JOLLY
"Threaded Interpretive Languages" H. T. GORDEN
"Microsoft BASIC Decoded" JAMES CARDELL

VOLUME 7, NUMBER 67, 1982

193 Letters
193 Editorial
197 Dr. Dobb's Clinic D. E. CORTESI
199 Of Interest MICHAEL WIESENBERG
202 Program for a Modem: Forth Code From the Conference Tree JOHN S. JAMES
208 North Star to Epson Connection LARRY SHOER
211 Add a System Clock MICHAEL KAN
217 New USER for CP/M GENE HEAD
223 Direct Video Output from the Sinclair ZX81 JERRY CHAMKIS
224 Using Computer Bulletin Boards MARLIN OUVERSON
226 Software Review
"COMMWHIZ: A Smart Terminal Package for TRS-80's" PCC STAFF
229 Book Reviews
"Interfacing to S-100/IEEE 696 Microcomputers" GEORGE W. JOLLY
"Lecture Notes in Computer Science" ROBERT IRVING
"Introduction to Computer Organization" GEORGE W. JOLLY
231 Hardware Review
"Hayes Stack Smartmodem" RICH HENDERSON-GRAGG
232 Guest Essay
"Prior Censorship" JOHN GILMORE

VOLUME 7, NUMBER 68, 1982

235 Letters
235 Editorial

237 Dr. Dobb's Clinic D. E. CORTESI
239 Of Interest MICHAEL WIESENBERG
241 PDP-1802—Hardware Simulator in Assembly Language
CHRISTOPHER L. HAMLIN
244 Multi-68000s in a Personal System AL KOSSOW
248 An Improved LET Processor for LLL Basic NORMAN E. THAGARD and GEORGE GLENN
256 A Print Utility for CP/M DAN DAETWYLER
266 A Cross-Reference Generator in C JEFFREY L. TAYLOR
271 Book Reviews
"Elementary Pascal" D. C. SHOEMAKER
"Pascal Programs for Scientists and Engineers"
WILLIAM G. HUTCHISON, JR.

VOLUME 7, NUMBER 69, 1982

275 Letters
275 Editorial
277 CP/M Exchange GENE HEAD
278 Dr. Dobb's Clinic D. E. CORTESI
280 CP/M-86 vs. MSDOS—A Technical Comparison DAVE CORTESI
292 Graphics on IBM's Personal Computer RAY DUNCAN
302 More on Checksums ROBERT BLUM
304 PDP-1802, Part 2—Hardware Simulator H. T. GORDON
310 Of Interest MICHAEL WIESENBERG
313 Software Reviews
"Volkswriter" ROBERT K. FUNKHOUSER
"Write-On!" JERRY CRIMMINS
316 Small-C for the 9900 MATTHEW HALFANT
323 Book Reviews
"Ciarcia's Circuit Cellar Volume II" GEORGE W. JOLLY
"Using Structured Design" CHUCK BALLINGER
"Computer Simulation in Business" ROBERT ASHWORTH

VOLUME 7, NUMBER 70, 1982

327 Letters
328 Editorial
329 Dr. Dobb's Clinic D. E. CORTESI
331 Argum—A "C" Command Line Processor ANTHONY SKJELLUM
348 Performance Evaluation in Small Computers MICHAEL KAN
355 SEND/RECEIVE File Transfer Utilities LAWRENCE HUGHES
361 Intel's 8087 Numeric Data Processor RAY DUNCAN
364 Of Interest

VOLUME 7, NUMBER 71, 1982

371 Letters
372 Editorial
373 Dr. Dobb's Clinic D. E. CORTESI
377 Forth Floating Point Package A. J. MONROE
386 Forth Programming Style N. SOLNTSEFF
388 Relocating, Linking Loader in Forth JOE BARNHART

393 H-19 Screen Editor in Forth A. S. WOODHULL
398 Z8000 Forth LOU ODETTE
413 16-Bit Software Toolbox RAY DUNCAN
415 Book Reviews
"Starting Forth" RICHARD L. LOZES
"Discover Forth" GEORGE W. JOLLY
"Programming Language Translation" EDWARD T. ORDMAN
417 Software Review
"The Working Forth Learning System" RAY DUNCAN
419 Of Interest MICHAEL WIESENBERG

VOLUME 7, NUMBER 72, 1982

423 Letters
423 Editorial
424 Dr. Dobb's Clinic DAVE CORTESI
427 68000 Cross Assembler ALLEN KOSSOW
439 The Portable Pidgin: Z80 Macro-Assembly Implementation
HERBERT GINTIS
449 Simplified 68000 Mnemonics W. D. MAURER
454 CP/M Exchange GENE HEAD
462 16-Bit Software Toolbox RAY DUNCAN
465 Of Interest MICHAEL WIESENBERG

VOLUME 7, NUMBER 73, 1982

469 CP/M Exchange GENE HEAD
469 Editorial
471 Dr. Dobb's Clinic DAVID CORTESI
472 Expand Wildcards Under UNIX ANTHONY SKJELLUM
476 Tests for Pidgin WILLIAM A. GALE
486 A Novice Builds the Sinclair ZX81 JOHN CONNELL, JR.
488 68000 Cross Assembler, Part 2 ALLEN KOSSOW
500 Memory Management in BASIC HENRY SEYMOUR
501 16-Bit Software Toolbox RAY DUNCAN
503 Software Review
"JRT Pascal" DAVE CORTESI
504 Book Review
"Teaching Your Computer To Talk" JOSEPH B. ROTHSTEIN
505 Of Interest MICHAEL WIESENBERG

VOLUME 7, NUMBER 74, 1982

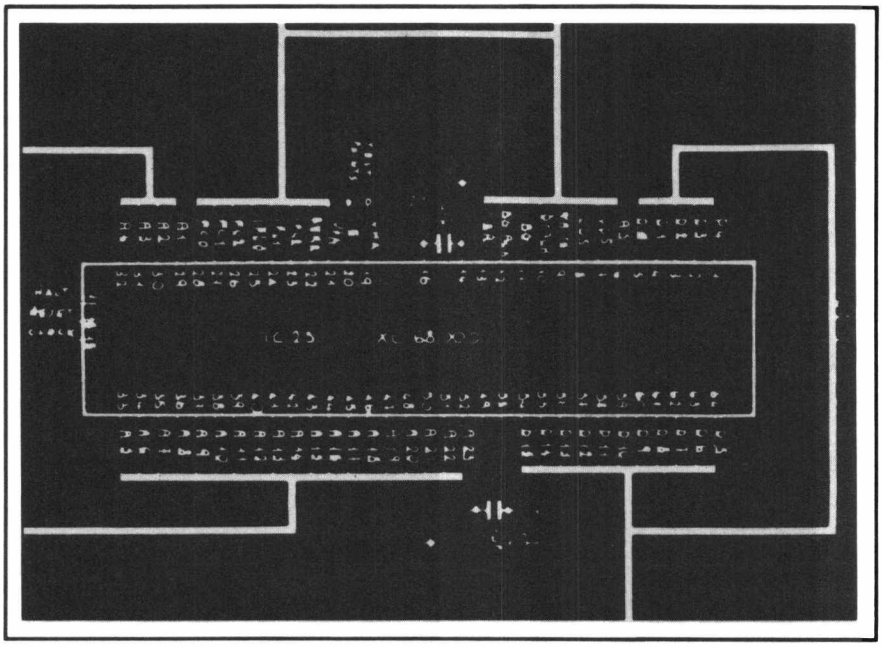
513 Letters
514 Editorial
515 16-Bit Software Toolbox RAY DUNCAN
520 Small-C Compiler, v.2 J. E. HENDRIX
546 Interrupts and CP/M ALAN BOMBERGER
550 A Simple Vector Generation Algorithm MAREK W. MICHALSKI
552 Guest Essay
"Fifth Generation Computers" RICHARD GRIGONIS
553 Dr. Dobb's Clinic JAN-HENRIK JOHANSSON
556 Of Interest MICHAEL WIESENBERG

$2.50
($3.00 in Canada)

DR. DOBB'S JOURNAL

For Users of Small Computer Systems

Number 63 January 1982 Volume 7, Issue 1



Interface the 68000 to AIM 65
Luca Fusina

A (Really!) Portable Screen-Oriented Editor
Edward K. Ream

Small Shell, Pt. II of a North Star Virtual Operating System
J.E. Hendrix

and even more!

Explore the 16-Bit World...

...Customize This Editor to Your Needs...

...Using "Co-routines" for Multiple Tasks

DR. DOBB'S JOURNAL
For Users of Small Computer Systems

January 1982 Volume 7, Issue 1, Number 63

CONTENTS

Interfacing the 68000 to an AIM 65 12
by Luca Fusina and Claudio Granuzzo

A Portable Screen-Oriented Editor 18
by Edward K. Ream

Small Shell: Part 2 of a North Star VOS 27
by J. E. Hendrix

Departments

Dr. Dobb's Clinic 4

Of Interest 10

Editor – Marlin Ouverson
Assistant Editor – Julie Anton
Contributing Editors – Dave Caulkins and Dave Cortesi
Marketing Director – Craig S. Harper
Advertising Sales – Laird Foshay
Circulation Assistants – Susan Hinton and Alice Hinton
Art Director – Clifford West
Production Manager – Barbara Ruzgerian
Typesetter – Renny Wiggins
Proofreader – Nancy Heubach

© 1981 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, Box E, 1263 El Camino Real, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025.

EDITORIAL

Nothing in Las Vegas was flashier or more fatiguing than the CLONEDEX convention last November. At least, that's what some were calling it. Four days was too much for anyone, even if there really were more than six hundred exhibitors. By the third afternoon, both tempers and the memorized sales pitches were getting shorter. That's when we left.

Communication at those events would be more open if there were a ban on three-piece armor. It is hard to trust someone so wrapped in corporate image that he takes off his individuality before donning his suit. Maybe Warren is right after all; roller skates and jeans require a kind of frankness that blue serge couldn't wring out of a saint.

If you are one of the thousands who didn't wear your soles thin, you missed demos of distributed processing/local networks, winchester-to-micro set-ups and some Visi-Clones (you'd have to check 'em all out yourself to discover the relative differences). I only have one question: where are all the geniuses who started this ball rolling? *They* surely aren't involved in making minor improvements on past achievements. We need some people thinking in terms of *light years*, not city blocks.

So, each of you, consider this a

CALL FOR INNOVATION

Think big, dream Bigger. Guest essays, dream pieces, signposts and other nefarious schemes for advancing this micro technology are welcome here, in addition to your hard software contributions to the community.

Now you have the whole story in a meager, encapsulated form. A few people were trying to figure why anyone, exhibitors or attendees, went to Las Vegas at all. I guess it was for the same reason people climb mountains – because it was there.

–MDO

Subscription Rates: $25 per year within the United States; $40 for first class to Canada and Mexico; $58 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

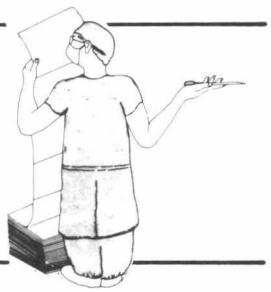
Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Eric H. Johnson, Jerry Horn, Edward W. Carr, Steven Fisher, Thomas H. Grohne, G. V. Elkins, Robert M. Connors, William J. McEown, Robert C. Luckey, Kim Leary, Robert N. Stabler, DeWitt S. Brown, John B. Palmer, E. Glion Curtis, Jack Miller, Burks A. Smith, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Howard E. Decker, Jim Wilson, Frank Lawyer, Chris Pettus. **Retaining Subscribers:** Osamu Shimizu, Scott B. Guthery, Robert L. Edens, Titus Purdin, Steven Fisher Enterprises. **Sustaining Subscribers:** Gordon Eubanks, Jr. and Pete Roberts. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** LP Enterprises, 8/11 Cambridge House, Cambridge Road, Barking, Essex, 1G11 8NT, England (Tel. 01-591-6511, TLX 892395). Hofacker-Verlag, Tegernseer Strass 18, D-8150 Holzkirchen, West Germany. **Asia & Austraila:** Computer Store, POB 31-261, 22B Milford Road, Milford, Auckland 9, New Zealand. ASCII Publishing Co., Ltd., Aoyama Building 5F, 5-16-1 Minami Aoyama, Minato-ku, Tokyo 107, Japan.

DR. DOBB'S CLINIC



The Matter of Coroutines

In the September Clinic we presented William A. McWorter's code for creating coroutine linkage between two programs. This month we have elaborations on that theme from two readers.

We checked Knuth's standard reference on the subject. He says:

"In contrast to the unsymmetric relationship between a main routine and a subroutine, there is complete symmetry between coroutines, which *call on each other* . . . . The essential difference between routine-subroutine and coroutine-coroutine linkage . . . is that a subroutine is always initiated *at its beginning*, i.e., at a fixed place, while the . . . coroutine is always initiated *at the place following* where it last terminated."[1]

Knuth attributes the coining of the term to M. E. Conway, who used it in an assembler in 1958 and who wrote the first published description of the structure in 1963[2].

Multiple Coroutines . . .

With that background, let us turn to a letter from David A. Shaw of Elmhurst, IL. He writes:

"The combination of coroutines and message queues makes for a powerful set of tools for data communication work. I have used them to implement a communications system on an 8080 system, based on a subset of High Level Data Link Control (HDLC), which has seven separate tasks executing in "round robin" fashion. I am currently designing a protocol converter that uses 4 coroutines, again running "round robin," and will be using them again for a full SNA/SDLC implementation on a Perkin-Elmer 3220.

"The enclosed listing [Listing I, page 4] is a slight modification of the 'SWAP' routine that I am using for the protocol converter. This version lets the calling task specify the next task to be started. For example, if 'TASK1' was running and wanted to pass control to 'TASK3', it would call 'SWAP' as follows:

```
MVI   A,T3.INDX    Name next routine
CALL  SWAP         Start 'TASK3'
```

"Anyone implementing coroutines on a system that uses interrupts should keep two things in mind. First, remember that interrupt service routines generally use the top of the user stack, so you should make each stack large enough to

Listing I

```
                                00002  *
                                00003  *       Multiple-Coroutine Stack Manager
                                00004  *
                                00005  *       Written by: David A. Shaw
                                00006  *
                                00007  *       This routine provides stack management services for up
                                00008  *       to 128 separate coroutines.  In this implementation,
                                00009  *       the routine that is giving up control names its
                                00010  *       successor routine.
                                00011  *
                                00012  *       Calling Sequence:
                                00013  *
                                00014  *               MVI     A,<coroutine stack index value>
                                00015  *               CALL    SWAP
                                00016  *
                                00017  *       The callers stack is preserved, but all registers can be
                                00018  *       assumed to be changed upon return.


                                00020  *       Stack Setup.
                                00021  *
                                00022  *       Each task has its own stack.  Each stack is pre-loaded
                                00023  *       with the first "Return" address for the task, which
                                00024  *       should be its startup and initialization routines.
                                00025  *
                                00026  *
000.031                         00027  STAKSIZ EQU     25              Allow 25 words on stack
                                00028  *
042.200                         00029          DS      (STAKSIZ-1)*2   Allocate most of stack
042.260 000 050                 00030  T1.STRT DW      T1ENTRY         Initialization entry for T1
042.262                         00031  T1STACK EQU     *               Actual stack address for T1
                                00032  *
042.262                         00033          DS      (STAKSIZ-1)*2
042.342 000 060                 00034  T2.STRT DW      T2ENTRY         Initialization entry for T2
042.344                         00035  T2STACK EQU     *
                                00036  *
042.344                         00037          DS      (STAKSIZ-1)*2
043.024 000 070                 00038  T3.STRT DW      T3ENTRY         Initialization entry for T3
043.026                         00039  T3STACK EQU     *


                                00041  *       Current Stack Address Table.
                                00042  *
                                00043  *       There should be an entry in this table for each task.
                                00044  *       The table is used to hold the current stack pointer
                                00045  *       while the task is "swapped".  I have also taken
                                00046  *       this opportunity to define index constants that
                                00047  *       should be used to "name" the next task to run.
                                00048  *
                                00049  *       More tasks can be handled by adding a stack
                                00050  *       and adding to the end of the Stack Address Table.
                                00051  *
                                00052  *
043.026                         00053  STAKTAB EQU     *
                                00054  *
000.000                         00055  T1.INDX EQU     *-STAKTAB       Index of T1 entry
043.026 260 042                 00056          DW      T1.STRT         Initial stack pointer
                                00057  *
000.002                         00058  T2.INDX EQU     *-STAKTAB       Index of T2 entry
043.030 342 042                 00059          DW      T2.STRT
                                00060  *
000.004                         00061  T3.INDX EQU     *-STAKTAB       Index of T3 entry
043.032 024 043                 00062          DW      T3.STRT
                                00063  *
                                00064  *
                                00065  *       Pointer to table entry for currently executing task.
                                00066  *       Should initially be set to the table entry of the
                                00067  *       first task to be run.
                                00068  *
043.034 026 043                 00069  CURSTAK DW      STAKTAB+T1.INDX


                                00071  *       Startup.
                                00072  *
                                00073  *       This routine enters the first coroutine.
                                00074  *
043.036                         00075  BEGIN   EQU     *
043.036 052 034 043             00076          LHLD    CURSTAK         Get first table address
```

(Listing I and text continued on page 6)

allow for interrupt service as well as the user task. Likewise, an interrupt can occur anywhere, so there should always be a valid stack pointer."

```
043.041 176              00077          MOV     A,M
043.042 043              00078          INX     H
043.043 146              00079          MOV     H,M
043.044 157              00080          MOV     I,A             (HL) = First stack pointer
043.045 371              00081          SPHL                    Load (SP)
043.046 311              00082          RET                     Start first coroutine


                         00084  *       SWAP    - Suspend processing of this task and start
                         00085  *                 the next one.
                         00086  *
                         00087  *       ENTRY   - (A)  = Index of STAKTAB entry of next
                         00088  *                         routine to be run
                         00089  *                 (SP) = Current task stack pointer
                         00090  *       EXIT    - (SP) = Next task stack pointer
                         00091  *                         Will "return" to next routine
                         00092  *       USES    - A,F,H,L,D,E
                         00093  *
043.047                  00094  SWAP    EQU     *
043.047 041 000 000      00095          LXI     H,0
043.052 071              00096          DAD     SP
043.053 353              00097          XCHG                    (DE) = Current routine stack
043.054 052 034 043      00098          LHLD    CURSTAK         (HL) = Addr of current table entry
043.057 163              00099          MOV     M,E             Put (SP) value in stack
043.060 043              00100          INX     H
043.061 162              00101          MOV     M,D
043.062 315 076 043      00102          CALL    GETADDR         (HL) = Address of next table entry
043.065 042 034 043      00103          SHLD    CURSTAK         Save address for next 'swap'
043.070 136              00104          MOV     E,M             (DE) = New routine (SP) value
043.071 043              00105          INX     H
043.072 126              00106          MOV     D,M
043.073 353              00107          XCHG
043.074 371              00108          SPHL                    (SP) = New routine stack
043.075 311              00109          RET                     Start next routine

                         00111  *       GETADDR - Get address of next stack table entry.
                         00112  *
                         00113  *       ENTRY   - (A)  = Index into table
                         00114  *       EXIT    - (HL) = Table entry address
                         00115  *       USES    - A,F,H,L
                         00116  *
043.076                  00117  GETADDR EQU     *
043.076 041 026 043      00118          LXI     H,STAKTAB       (HL) = Addr of front of table
043.101 205              00119          ADD     L               Add subscript value
043.102 157              00120          MOV     L,A
043.103 320              00121          RNC                     No carry - all done
043.104 044              00122          INR     H               Else bump high byte
043.105 311              00123          RET


                         00125  *       Define some dummy task starting addresses,
                         00126  *       so this assembly will be clean.
                         00127  *
050.000                  00128  T1ENTRY EQU     50000A
060.000                  00129  T2ENTRY EQU     60000A
070.000                  00130  T3ENTRY EQU     70000A
                         00131  *
043.106 000              00132          END     BEGIN
```

End Listing

. . . and Multiple Tasks

The coroutine is a powerful tool for structuring some applications. Others, especially those that have to respond to unpredictable events, fall out better as completely independent tasks. The smallest operating system that supports multitasking (that we know of) is Digital Research's MP/M, and it is not effective on a typical personal machine. If we want to design a program as a set of cooperating tasks, we have to supply the task-dispatching mechansim as part of the program.

That is exactly what A. James Laurino of Philadelphia, PA did. He wrote to tell about it:

"It is important to note that multitasking and similar approaches can tremendously simplify some very hard problems. Many people shy away from multitasking because they feel it is too difficult to implement. Thus they lock themselves out of a lot of worthwhile uses for their machines that are too complex without some reliable structuring method.

"I want to dispel some of the mystery around multi-tasking by showing a simple kernel that I have evolved over the last few years. The latest version was written for the 6800 and is only forty-one instructions long. Dijkstra's semaphore mechanism is used as the synchronization method. Briefly, a semaphore is a counter whose value represents the difference between the number of times an event has occurred and the number of tasks that are waiting for the event. In particular, a negative count indicates that one or more tasks are waiting for the associated event.

"In this realization a maximum of seven concurrent tasks was more than adequate. Each task has an associated priority, and tasks receive service in priority order. A task is either ready to run or else it is blocked, waiting for some event. The implementation also allows for one level of interrupt and for a background (non-blocking) task.

"We use a series of queues to keep track of tasks. There is one queue that holds all the tasks that are ready to run, and there is one queue associated with each semaphore that holds all the tasks that are waiting for that event. Each queue is kept in priority order, of course.

"A semaphore count is bounded by the precision of the counter used to model it. In this design a maximum count of +127 was more than adequate. The negative count of a semaphore is bounded by the number of tasks in the system. One byte thus suffices for each semaphore count. A queue also has a maximum size bounded by the number of tasks in the system. We can achieve a considerable reduction in the code and storage requirements over conventional designs by representing each task by a single bit position in a byte that represents a queue. The sign bit is set to indicate that the byte contains a queue, and each other bit that is set indicates that the associated task is in that queue. The semaphore queue can be the same byte that holds the semaphore count. When the count is positive, the queue is unused. When the count is negative, we represent the count by the number of bits set in the queue.

"There are two calls to manipulate semaphores, *signal* and *wait*. A *signal* indicates that an event has occurred; the semaphore count must be increased by one and, if the count was negative, the highest priority task waiting in the queue must be moved to the ready queue. The *wait* is used to synchronize a task with an event outside it; the semaphore count

must be decremented, and if the result is negative, the calling task must be removed from the ready queue and placed in the semaphore queue. I will return to the subject of queue manipulation after I have introduced the context switch mechanism.

"By context switch I mean the process of allocating the processor to a new task. Interrupts perform this trick all the time, of course. In an interrupt, the processor registers are saved on a stack and control is transferred to the interrupt routine. When the interrupt routine is done the old register values are restored from the stack and the interrupted routine resumes processing. In software multitasking, each task is assigned a stack of its own. After the task's registers have been saved we also save the stack pointer. We decide which task is to run next by examining the ready queue. We load the saved stack pointer of the highest priority task in the queue, and then reload the processor registers from that stack.

"Queue manipulation can be accomplished by simple, parallel operations on the queue bytes if we observe the convention that the highest priority task is assigned to the rightmost bit position. The rightmost bit in a word can be isolated by ANDing the word with its two's complement. Once it is isolated, we remove the bit from the queue by XORing it against the queue byte. We place the bit into a new queue by ORing the bit into the queue byte. The only time that the bit actually has to be identified is when we want to find the associated saved stack pointer. The example code [Listing II, page 7] does this by counting and shifting the ready queue until a bit arrives at the carry position.

"Storage requirements are modest. Each task must have a stack, but 20 or 30 bytes is usually sufficient. One byte is needed for every semaphore used. The kernel has a few private storage areas. One double byte is kept for each task to save its stack pointer. One byte is used for the ready queue. And a double byte is used to keep a pointer to the SP save area of the currently executing task to speed up the context switch.

"Semaphore operations are invoked by a software interrupt; in the 6800, this conveniently saves all the registers on the stack [the comparable 8080 operation, is RST; it does not save all the registers]. Hardware interrupts can call the *signal* mechanism as well; they are treated identically to an entry from an SWL instruction. The kernel expects to find the operation type and the semaphore address stored at the calling location. A typical call would look like this:

```
        SWI             ; pass control to
                          the kernel
```

(Continued on page 33)

Listing II

Multi-Tasking Kernel for 6800 Microprocesor

```
ENTRY:  TSX             ; task entry point. Note interrupts are off.
        LDX     5,X     ; reach back to call to get arguments.
        ASR     2,X     ; test function code. Note 0 and -1 won't change.
        LDX     3,X     ; get semaphore pointer into X.
        BCS     SIGNAL  ; 0 means Wait, leaves carry clear after ASR.

WAIT:   LDAA    0,X     ; get and test the semaphore value.
        BLE     BLOCK   ; 0 or less means this task must wait for event.
        DEC     0,X     ; else, just decrease the count
        RTI             ;   and continue in the same task.

BLOCK:  LDAB    READY   ; suspend the current task. Get bit of highest
        NEGB            ;   priority task on Ready queue (calling task).
        ANDB    READY   ;
        ABA             ; place this task bit in the semaphore queue.
        ORAA    #$80    ; make sure sign bit is set to mark as a queue.
        BRA     DISP    ; go find a new task to run, finish update.

SIGNAL: LDAA    0,X     ; interrupt entry point. Note interrupts are off.
        BLT     UNBLOK  ; if there are tasks on sempahore queue, unblock one.
        INC     0,X     ; else, just count the event.
        BVC     .+3     ; make sure the count can never overflow.
        DEC     0,X     ; keep at max value in case of too many events.
        RTI             ; continue in the same task as before.

UNBLOK: NEGA            ; activate highest priority task in the current queue.
        ANDA    0,X     ; get highest priority task bit in the queue.
        TAB             ; save for Ready update below.
        EORA    0,X     ; remove the bit from the semaphore queue.
        CMPA    #$80    ; test for an empty queue.
        BNE     DISP    ; continue if not empty.
        CLRA            ; else, turn the queue back into a counter again.

DISP:   STAA    0,X     ; run the highest priority task in the ready queue.
        EORB    READY   ; but first update Ready queue. B has bit to add
        STAB    READY   ;   or to remove. Also save updated semaphore value.
        LDX     CURNT   ; get pointer to SP save area for the current task.
        STS     0,X     ; save the stack pointer for the current task.
        LDX     #SPSAV-2 ; set up to generate pointer to SP area of new task.
LOOP:   INX             ; each pass through loop advances X to next SP area.
        INX             ; note SP areas are ordered highest priority first.
        LSRB            ; put right bit into carry, ready status of next task.
        BCC     LOOP    ; try next highest if this one not ready.
        STX     CURNT   ; found one to run. Save copy of SP area pointer.
        LDS     0,X     ; now reload hardware SP from this save area.
        RTI             ; reload registers and resume running this task.
```

End Listing

NOTES: Interrupts store registers in order: flags, B, A, XH, XL, PCH, PCL. A non-blocking task must exit or the system will hang. If a dummy task is used (BRA .), its stack will never change and can be in ROM. Minimum stack is 7 bytes. Transparent call which preserves all registers allows reliable use of macros for semaphore operations.

OF INTEREST

Impressive Info

Our attention was caught this month by pamphlets from two vendors. **Torres Ribbon Service** has a free brochure designed to make us more informed consumers of film ribbons for our printer. The brochure is said to explain different coating processes, the resulting variations in impression quality, and other matters. And **Graphic Dimensions** offers a $6.95 booklet, "Getting the Type You Want," which is said to be "especially helpful for word processor operators who are baffled by the technical jargon used in the typesetting industry."

A PCC Test Report

We received an "ABS Doubler" free from **ABS Suppliers.** It's a slip of stiff plastic with a hole in it, said to be "a precision template to permit the conversion of single-side diskettes to 'flippies' or double-side diskettes." The price of $3.50 seemed steep for something the size of a playing card, so the Editor was dragooned into testing it. Here is his report:

"The template worked, but takes a very steady hand to insert a paper punch between the disk and its jacket without scratching the surface. With reasonable care it can be done, and at the cost of disks these days, it may be well worth the wracked nerves. Two ordinary diskettes were converted and both worked as 'flippies' on a TRS-80 III. No 'bleed-through' effect was evident."

What Price an Atari?

The influential Minnesota Educational Computing Consortium (MECC) sent a newsletter that tells of its award to Atari Inc. of a one-year contract to supply computers to Minnesota schools. Atari, MECC says, will supply Atari 400 computer systems for $575. That $257 for the 400 console (RAM not specified), $243 for an 810 disk drive, and $75 for a black-and-white monitor. For comparison, the mail-order houses are currently quoting the 400 at $350 and the 810 at $450. What makes the MECC prices even more remarkable is that the system will be sold through Atari dealers in Minnesota. We can imagine the factory's taking a loss on a big sale in exchange for the prestige and exposure that it will bring. But we can't see a local dealer doing it, so those prices are probably close to dealer cost. It's of interest, isn't it, when a small disk system costs less than a good business suit?

Microsoft Moves

Microsoft, Inc., a company you may have heard of, wants you to know that they have moved. Note the new address and phone number under *Contact Points.*

Your 1982 Calendar

Here are some conferences that you might like to attend this year. Start with January 28-29 in San Diego, at the **Society for Computer Simulation**'s conference on Modeling and Simulation on Microcomputers. Stay in the sunbelt to swing east to Orlando, Florida, for **Computer Expo '82** on February 26-28. It's a show for the general public, sponsored by Adventure International.

Educators will be well-served with conferences throughout the rest of the spring. **MACUL '82** is a confab put on in Kalamazoo, March 1-2, by the Michigan Association for Computer Users. Commander Grace M. Hopper will be the featured speaker.

Fly out the night of the second to Jersey City, NJ; you'll just make Microcomputer Week '82, March 3-7, sponsored by **CATALYST**. This, to judge from a packed, 5-page release, will be *the* computer education conference of the year, featuring "150 leaders in the field" (are there so many?), 66 short courses with hands-on labs, and many general sessions and discussion groups.

The Alaska Association for Computers in Education **(AACE)** will be holding its 1982 conference in Anchorage, April 1-3, at which "heavy attendance . . . from all over the state" is expected. It may not be too late to reserve a vendor booth when you read this. The Computer-Using Educators **(CUE)** will hold their 1982 southern conference a month later, May 14-15 in Irvine, CA.

Graphics and Robotics persons can, if they hustle, still submit a paper to the **IEEE**'s 1982 conference on Pattern Recognition and Image Processing. That is planned for June 13-17 in Anaheim, CA.

The big conference outside of the US will be put on by **Euromicro**, September 5-9 in Haifa, Israel. This one is a must for those planning an assault on the European market. Submit papers to **C. J. van Spronsen.**

Comal and Pilot

"COMAL" is a word that will be heard often at Euromicro '82. COMAL (described recently in these pages) is being pushed as a better replacement for BASIC. The **COMAL Users Group**, in a transport of enthusiasm, claim it "gives you the ease of BASIC, along with the power of Pascal." Hmmm. If you have a CBM machine with a disk, you can order their COMAL starter kit for $47.50 plus shipping and find out for yourself. It includes the interpreter, a manual, and a batch of sample programs. Or send a large envelope with 40 cents postage for an information packet.

"Pilot" will doubtless be heard often at the US education conferences. It is usually touted as a simple language with which anyone can write CAI courses. A company called the International Institute of Applied Technology **(IIAT)**, has prepared a $150, machine-independent, implementation of Pilot. They have versions ready for most Z80 machines, and plan to add the IBM Personal Computer and some 6502 systems in 1982.

Another outfit, **Stok Computer Interface**, has a version of Pilot for CP/M that they claim is good for CP/M system control as well as for CAI courses. Stok Pilot can be either interpreted or compiled into a .COM file. This $109 language, they say, can be used to write menu interfaces to any system of CP/M commands.

The N/C Apple

"N/C" is an acronym for "Numerical Control," the term for driving an industrial milling machine or lathe through a sequence of simple movement instructions expressed as numeric codes. The tool program used to be coded in paper tape, but we suppose that machines have gotten smarter these days. The problem of translating a complicated milling operation from human terms into detailed motion codes was attacked with computers almost as soon as there were computers; N/C languages were among the first problem-oriented computer languages. They've traditionally been run from time-sharing terminals, but now **Arrow Data Systems** has build N/C software for the Apple. With an Apple and this $1795 package, they claim, "even the smallest N/C shop" can prepare N/C tapes with a computer.

The Bisync Z80

Lifeboat is now offering RBTE-80, a CP/M-based emulator for the IBM batch workstation terminals. Like Radio Shack's similar product for the Model II, this package uses the IBM 2780 protocol to transfer files between the small system and any of a number of makes of mainframes. That lets the small machine's owner upload jobs for the mainframe to do, and download the resulting datasets.

Two things bother us about this announcement. The $750 price seems stiff, even for a full-feature product with diagnostics and line trace facilities. But if that judgement is right, the market will take care of the problem. There are plenty of people with the telecommunications expertise to produce competitive products.

In the longer run, it disturbs us to see what looks like a Bisync Bandwagon beginning to roll. The Bisync protocol is an old architecture. It is just adequate for communication on land lines; it falls flat when the long delays inherent in satellite communications are added. Yet the much more efficient, flexible, and reliable X25 protocol and its near relatives, IBM's SDLC and DEC's HDLC, are being ignored. That, too, may be an effect of the marketplace, but it's too bad.

The Video TI

Texas Instruments has announced a Video Controller that lets the 99/4A computer control some video tapes and the Pioneer video disc. With the accompanying Course Designer software, the user can write a course that combines computer displays, segments of video material, and synthesized speech. The course software can be stored on the second audio track of the video medium. This sounds like an impressive package, one that will permit the creation of educational material of terrific quality from very reasonably priced equipment. The Video Controller is priced at $700, and the software costs $200.

The 1.5MB H89

Hard on the heels of Heath's announcement of quad-density disks for the H/Z89 computer came an announcement from **Percom**. They have prepared a 4-drive controller card for their double-density 5" drives for the H89. The $250 card contains an extra, parallel port. It does not replace the existing Heath controller. Instead, Percom claims that their controller and the regular one work in tandem. They say that any kind of diskette may be inserted into any drive, and their software patch will select the correct controller to read it.

The Sensitive Apple

Strawberry Tree Computer have designed a thermometer card for the Apple II. The $260 card contains two independent thermometers, each with a 10-foot probe, each measuring from −55 to +125 degrees C. Accompanying software takes, records, and displays the measurements, and can handle up to seven cards. Sounds like just the thing for monitoring a solar-power system, among other things.

The Tall S-100

Inner Access Corp. has announced a couple of S-100 boards that might interest hardware folk. They have a prototype board in the new, 10" height. It will accommodate 112 ICs and has regulators for the 5- and 12-volt supplies. They also have a 13-inch extender card (needed to get above the height of a row of 10-inch boards). The prototype board is $120, the extender is $58.

In the same vein, here's a product idea for Inner Access or any other hardware outfit with S-100 know-how. Our system went out with bus problems recently, and while trying to diagnose it we thought of the ideal diagnostic tool. That would be a pair of S-100 boards joined by a cable. One board would plug into the busted bus to buffer and latch its signals. The other would plug into a working system and present the state of the machine under test to diagnostic software through a set of I/O ports. Proper software could find a lot of bus problems, or even drive the test system in the manner of an in-circuit emulator. Send along the announcement as soon as you've got it working, please. ▶▶J

Contact Points

AACE '82 Conference, 1602 Hillcrest Drive, Anchorage, AK 99503; exhibitors call Pat Stowers, (907) 683-2278.

ABS Suppliers, POB 8297, Ann Arbor, MI 48107; (313) 971-1404.

Arrow Data Systems, 5910 East Washington, Los Angeles, CA 90040; (213) 726-9440.

CATALYST Conference, H112, Jersey City State College, 2039 Kennedy Boulevard, Jersey City, NJ 07305; (201) 434-2154 or (201) 547-3094.

COMAL Users Group, 5501 Groveland Terrace, Madison, WI 53716.

Computer Expo '82, Tom Blayney, Executive Director, POB 1185, Longwood, FL 32750; (305) 339-1731.

CUE Computers in Education '82, Craig Walker chairperson, Arrowview Intermediate School, 2299 North "G" Street, San Bernadino, CA 92405; (714) 886-9118.

Euromicro 1982 Conference, 4 Place Félix Eboué, 75012 Paris, France; (1)341.08.46; telex 211801.

Graphic Dimensions, 8 Fredrick Road, Pittsford, NY 14534.

IEEE Computer Society, 1982 Conference on Pattern Recognition, POB 639, Silver Spring, MD 20901; (301) 589-3386. Submit papers to Dr. Richard P. Kruger, Los Alamos National Laboratory, Los Alamos, NM 87545.

IIAT, 20010 Century Boulevard, Germantown, MD 20767; (301) 428-9010.

Inner Access Corp., 517-K Marine View, Belmont, CA 94002; (415) 591-8295.

Lifeboat Associates, 1651 Third Avenue, New York, NY 10028; (212) 860-0300.

MACUL '82, c/o Carolyn Gilbreath, Oakland Schools, 2100 Pontiac Lake Road, Pontiac, MI 48054; (313) 858-1898.

Microsoft, Inc., 10070 Northup Way, Bellevue, WA 98004; (206) 828-8080.

Percom, 11220 Pagemill Road, Dallas, TX 75243; (214) 340-7081.

Society for Computer Simulation (SCS), POB 2228, La Jolla, CA 92038.

C. J. van Spronsen, Euromicro Program Chairman, Delft University of Technology, P. O. Box 5031, 2600 GA Delft, The Netherlands; (31)(15)786227; telex 38070.

Stok Computer Interface, POB 501, Woodside, NY 11377; (212) 426-7022.

Strawberry Tree Computers, 949 Cascade Drive, Sunnyvale, CA 94087; (408) 736-3083.

Texas Instruments, Customer Relations, POB 53, Lubbock, TX 79408.

Torres Ribbon Service, POB 1727, Redlands, CA 92373.

Interfacing the 68000 to an AIM 65

Today there are several 16-bit CPUs on the market; for a number of reasons the most famous are only three, the 8086, the Z8000 and the Motorola MC68000. We have been hearing a lot about what they can and cannot do; as a matter of fact, boards employing one of such CPUs did not have too much luck.

This recalls to memory what happened years ago: everybody realized that the Z80 was more powerful than the 8080, but an upgrading kit for 8080 machines simply did not sell. The reason was not the lack of Z80 software, because the Z80 can run 8080 software. It was that the user was not willing to spend money for something maybe better but not "useful."

Nowadays we have to ask ourselves how useful is a 16-bit CPU; we feel it is not for the computer consumer, the one who buys a computer just to play games or little more. There is a range of applications requiring more computational power than what is currently available on 8-bit CPUs. When we say "computational power," we do not only mean an extended instruction set or the capability of running standard programs ten times faster: all those aspects have to be considered as a whole, along with all the hardware facilities. Many concepts developed for the old mainframes are becoming prominent in microsystem design: can you imagine a multi-task, multiprocessor system without the test-and-set instruction and the user-supervisor environment? Even the hobbyist with little background can successfully experiment with concurrent programming on a fairly small system, provided he has the right CPU to start with.

We think that a good 16-bit CPU has enough power to handle fairly sophisticated, concurrent programming. Among the available devices, the MC68000 is a good choice. Besides its nice hardware structure, the following points are to be taken into account:

1) Its instructions are powerful but limited in number. We feel that a large instruction set does not necessarily make a processor more powerful, it may instead confuse the programmer.
2) The instruction set is microprogrammed; it may be improved without changing the overall structure.
3) Its pipelined structure is optimized for speed.
4) It is asynchronous; this feature allows for easy interfacing with all kinds of devices and peripherals.

The 68000 has some drawbacks, too:

1) It does not provide a dynamic memory refresh like the Z8000; this is a really handy feature, even if software refresh is an alternative.
2) Its most interesting supporting chips are not scheduled to be available in the near future.
3) Like most of its competitors, there is not too much software available if we exclude that delivered from Motorola for their boards and development systems (which, in any case, are quite expensive).

Luca Fusina and Claudio Granuzzo

Luca Fusina, Via Mocenigo 8, Verona 37100, Italy.

All the article content may be used for any non-commercial purpose.

Nowadays it seems that what really interests the computer consumer is software. He does not care too much about the underlying hardware, he is most concerned about the programs he can run on his machine. At most he considers the interface capabilities of his computer, but usually he even does not know the internal hardware structure. In our opinion there are still some people interested in system design: people who like to experiment with new devices. The tool for this kind of work is called a development system. After the hobbyist eliminates the ones that are too expensive, what is left is an evaluation system which basically is a single board equipped with a CPU and a handful of switches and LEDs. Experimenting with such boards is time consuming and not rewarding at all in any case. There must be a better solution.

The Idea

Almost all of the computer enthusiasts around already own a microcomputer. Why not use our own system to control a 68000? In this article we will describe this hardware and software implementation on a Rockwell AIM 65. For the reader who is unfamiliar with this machine we will summarize its features. It is a single board based on the 6502 with a QWERTY keyboard, a 20-column thermal printer and an alphanumeric display, 4K byte of RAM, an 8K byte ROM monitor, a 4K byte ROM assembler, and 2 Versatile Interface Adapter chips for I/O and expansion. We had to make only a few trivial hardware changes.

We have said, indeed, that we are going to control our 68000, but which way? We want to be able to control it during each read or write cycle (from now on simply cycle), in real time, changing the mode of operation according to necessity among all the ones available to our system. Furthermore, we want to be able to manipulate during each cycle all the 68000 control signals; this way we can simulate interrupts, multi-processing and so on.

The Hardware

All we needed to implement our idea was nine eight-bit I/O ports and a couple of decoders. The 68000 became a peripheral connected to the AIM bus at a certain address. On the AIM expansion connector we found all we needed: the 8-bit bi-directional data lines, the 16-bit address lines, and two control signals: R/W and 02 which are used for synchronizing R/W operations. Figure 1 (page 15) shows the block diagram of the interface; Figure 2 (page 16) is the schematic. For all the 6502 timings, the reader can refer to the 6502 hardware manual.

From now on we assume the reader understands 68000 hardware and software details; refer to the MC68000 user's manual for a complete description of this processor.

As the 68000 is asynchronous, we can control each cycle using the signal $\overline{\text{DTACK}}$, which stands for data transfer acknowledge. When the 68000 wants to perform a read or a write, it asserts the $\overline{\text{AS}}$ signal and waits for $\overline{\text{DTACK}}$. At this point our AIM program can do all it has to, and when finished it asserts $\overline{\text{DTACK}}$ by writing to location $8XXB, signaling the 68000 to continue. IC 20 (a 74LS74 flip-flop) takes care of negating $\overline{\text{DTACK}}$ when 68000 negates $\overline{\text{AS}}$.

All 68000 signals are interfaced to the AIM through six DM81LS95 buffers and three 74LS373 latches. The 68000 interface is seen by the AIM as a 1Kbyte memory. The 6502

can address up to sixty-four 1Kbyte pages and the board can be located anywhere inside them, provided there are no conflicts with the AIM 65 requirements. This is accomplished with a DM8131 six-bit comparator (IC 10). To select the right page the user must supply the appropriate logic levels to the comparator inputs. In our implementation the board is located at $8000. Inside this address space there are sixteen meaningful locations, from address $8XX0 to address $8XXF. Only twelve are, however, actually used and only the first nine are used to interface the 68000, by means of selecting one of the I/O ports. Selection 10 (signal $\overline{Y9}$ in Figure 2, address $8XX9) is used to control the $\overline{HALT}$ line. Number eleven is not connected. Number twelve asserts $\overline{DTACK}$ (signal $\overline{Y11}$ in Figure 2, address $8XXB). Refer to Figures number 3, 4, 5 (page 17) for a detailed explanation about how the ports are arranged. Note that with this hardware it is not possible to use the M command of the AIM monitor inside the board space.

There are two 68000 signals not connected to any port: $\overline{VMA}$ (valid memory address) and E (enable). Motorola implemented these signals to maintain compatibility with existing 6800 peripherals. In our opinion it is better to use them as test points. The user can easily add one additional port to the interface to monitor them under program control.

The network of open collector inverting gates connected to the 555 timer and the $8XX9 selection is used to interface correctly the 68000 $\overline{HALT}$ and $\overline{RESET}$ signals, which are bidirectional. Two LEDs can be used to monitor their logic levels. The board performs an automatic power-on reset; a manual reset is also provided, as well as software $\overline{HALT}$ and $\overline{RESET}$ control.

The Software

Before entering into program details, it may be a good idea to lay down its objectives:

Run-time control of the dynamic evolution of each 68000 instruction step using the AIM keyboard. It is possible to execute a program stopping the 68000 every cycle, or to execute any number of instructions consecutively; a dynamic switch between these two modes is possible, too.

Run-time control of the 68000 input control signal (IC 19, #74LS374 latch). The AIM 65 keyboard can be used to supply a byte to be stored in that latch. We called this feature "dummy memory."

Output on display/printer of each 68000 cycle including addresses, data and control signals coming out from the processor.

Dynamic allocation of memory. 68000 memory is segmented and each segment base address can be located anywhere inside the AIM free RAM. If the 68000 is doing a read, it is possible to enter data from keyboard; if it is doing a write, it is possible to do a data display. This way no effective memory operations are done.

Use of all AIM 65 peripherals and utilities under 68000 program control. We defined that a 68000 segment cannot be greater than 64Kbytes (in automatic mode). If a write is performed inside the first 256 bytes of the last 1Kbyte page available to a segment (address $00FCXX) it is assumed that a 6502 subroutine call is made. The 68000 lower data byte is loaded in the 6502 accumulator. The 68000 upper data byte is used to index a table of pointers to 6502 subroutines. On return, the 6502 accumulator is copied into two locations, one inside the 68000 user data segment and the other inside its supervisor data segment.

The 68000 is operated cycle by cycle by the program in Listing 1, p. 36. As the program currently running evolves, whatever happens is shown on the display/printer or whatever is connected to the AIM 65. Various instruction combinations can be tried, and the operation of the 68000 becomes clear.

The five objectives that we have so far discussed are implemented in a program 519 bytes long. The user has to select the operating mode by storing an appropriate value in a control byte, CONTRL. Each bit controls a mode as explained below.

If bit 0 is set, after printing the 68000 address the AIM 65 program requests a byte from the keyboard. It will be stored in the latch connected to the 68000 input control signals.

If bit 1 is set, manual mode is selected, otherwise automatic mode is assumed. Manual mode corresponds to the dummy memory R/W mode previously discussed; in the auto mode, R/W is performed from memory.

If bit 2 is set, before issuing the $\overline{DTACK}$ signal the user is requested to validate all operations performed during the current cycle by entering a carriage return. This is also called step mode. Any other character will repeat the current cycle. No such validation check is made if bit 2 is cleared.

If bit 4 is set, fast mode is selected. When this mode is selected 68000 cycle status output is suppressed. This allows for fast 68000 program running, as a great deal of the AIM housekeeping time is spent reporting things on display/printer.

The previous modes can be mixed together. Fast mode overrides all the others. When it is set, the other modes are used to select the mode that will be entered upon error. Dynamic mode changing can be accomplished by executing a 68000 instruction that stores a new value in the control variable. Thus, a 68000 program can put itself in Step mode.

Memory segmented is implemented using two tables, FCTAB and MAXADD. They specify the AIM 65 base addresses of each 68000 segment and their extension. These latter values must be supplied, and they must be consistent with the former ones. Of course, each 68000 segment starts from location $0000000.

Final Thoughts

We have so far discussed how to build from scratch a small development system for Motorola's 16-bit processor, the MC68000. The underlying concepts are quite general, and it should not be difficult to implement our idea using the reader's own computer instead of our AIM 65. The ones familiar with this machine will have already noted that the accompanying program was not listed with the 20-column thermal printer available on the AIM board, and that the assembler was modified.

In fact, besides using the AIM 65 as an experimental computer, we use it as our "big" system. Anyone who is interested and wants more information about how we did it may write to the authors. Let us now report the impressions about the MC68000 that we gained using the board we have here presented. We appreciated most the power and simplicity of its instruction set. Some things shocked us, though. For example, you can try to execute a CLR to memory and see what happens. Before clearing the desired locations, the processor reads them. This, of course, wastes time and has no usefulness. Any-

way, summing up all the againsts and fors, it proved to be a superior processor. After executing just a few programs, the user accustomed to 8-bit microprocessors will no longer be satisfied with them.

The MC68000 architecture is a bit different from standard 8-bit machines. The reader who will use our board might then have some problems. Let us give some hints that may prove useful.

The 68000 has a pipelined structure. This means that it fetches one or more words before actually executing the instruction. These words, which may be subsequent instructions, are printed and displayed. Such a thing may lead one to think that the processor is not executing properly; on the contrary, it is doing its job. Again, during stacking operations, words may not be stacked following the address ordering; the overall stacking procedure is still correct.

The MC68000 is a 16-bit machine, therefore it addresses by words. Instructions must start on even boundaries. If the user specifies the initial PC to be at an odd address, the 68000 will enter an address error exception. If the supervisor stack point also starts at an odd address, well, you will be in trouble. There are, of course, many other things that should be said, but it may be more interesting to explore the 68000 world yourself.

Concluding, we would like to point out that everything we have said so far is not restricted to a particular machine. Some readers will like to experiment with a different CPU, say the Z8000, and we think they will not have too many problems adapting our ideas to their needs. After a few weeks of experimenting with our board, the need for more sophisticated software may arise. It should not be too difficult to write a cross assembler using AIM BASIC. This would eliminate the need for hand compiling all the 68000 instructions. Again, it is possible to slightly modify the hardware to let the 68000 have an independent life, without passing through the AIM for executing its instructions.

Italy is not that far away; anyone who wants to write us to exchange opinions about computers is encouraged to do so.

DDJ

(Figures 2-6 on pages 16 and 17)
(Listing begins on page 36)

68000 Control Board Block Schematics

by Luca Fusina and Claudio Granuzzo

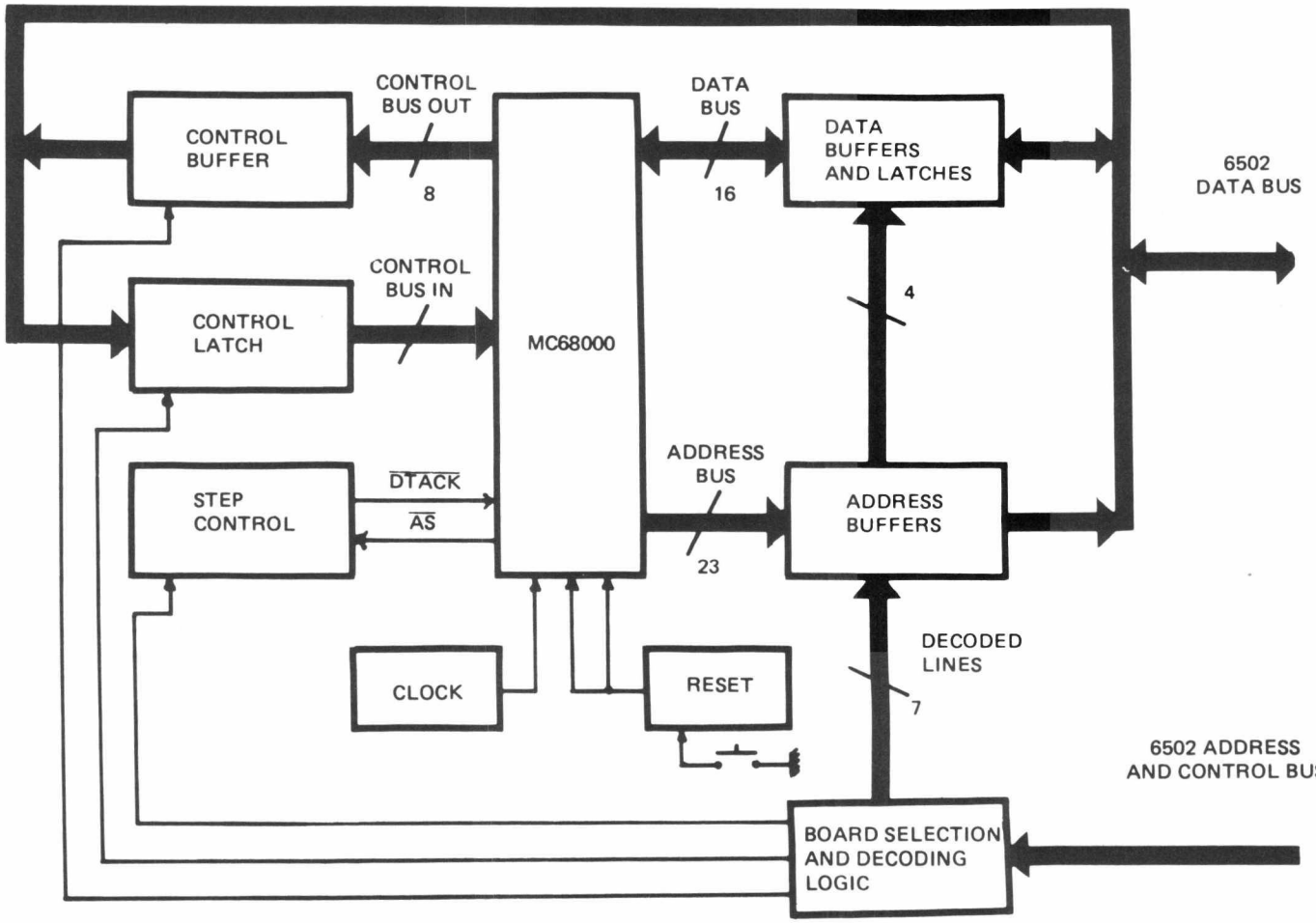


Figure 1

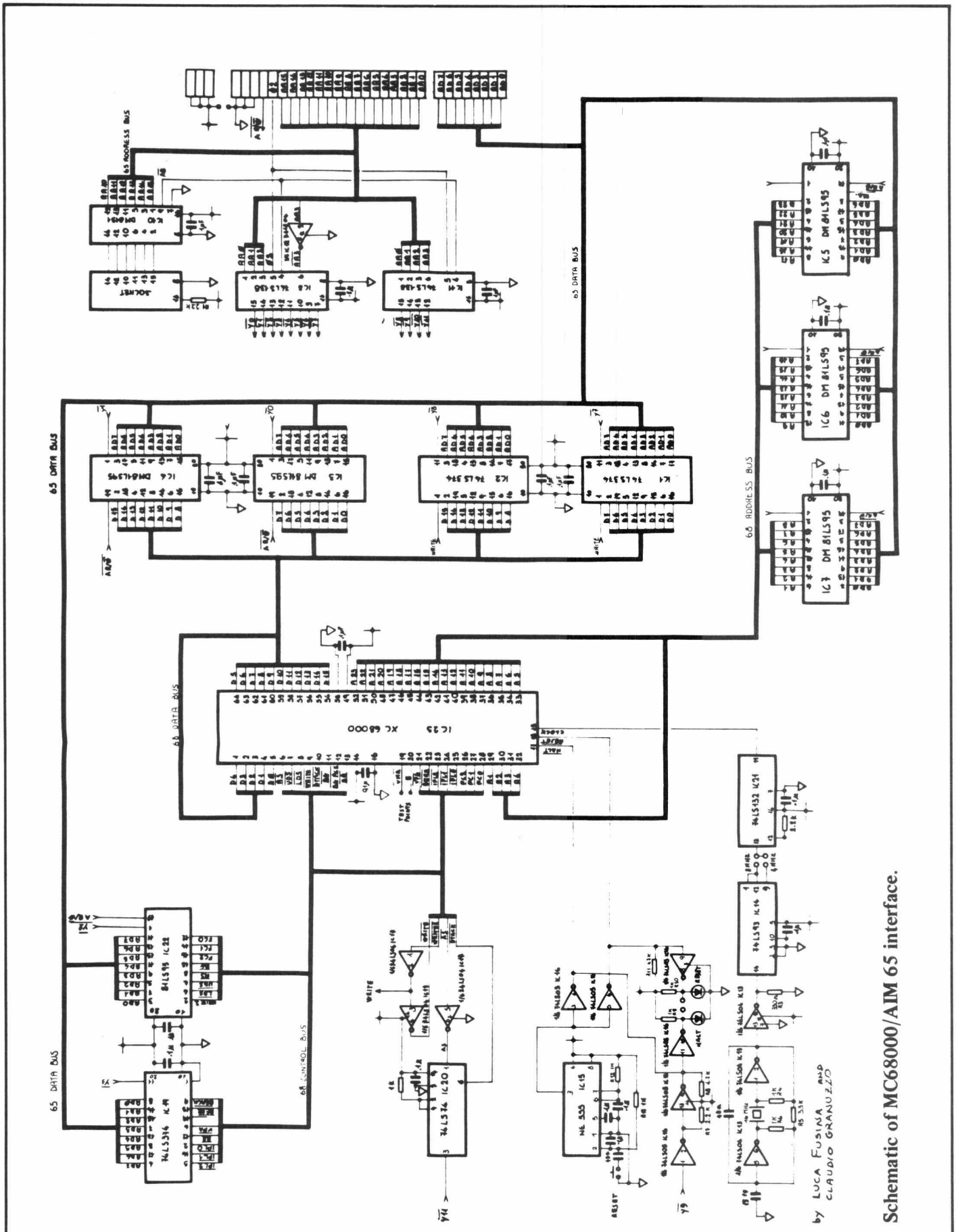

65 DATA BUS
65 ADDRESS BUS
68 DATA BUS
68 CONTROL BUS
IC 23 XC 68000
IC 20
IC 22
IC 21
IC 16
IC 15
IC 1 DM 81LS95
IC 6 DM 81LS95
IC 5 DM 81LS95
IC 4 DM81LS95
K1 74LS374
K2 74LS374
K3 DM 81LS95
K4 74LS374
K5 74LS374
K6 74LS374
K7 74LS374
K8 74LS374
K9 74LS374
K10 74LS374
K11 74LS374
K12 74LS374
K13 74LS374
K14 74LS374
K15 74LS374
K16 74LS374
K17 74LS374
K18 74LS374
K19 74LS374
K20 74LS374
K21 74LS32
K22 74LS08
K23 74LS08
K24 74LS08
K25 74LS08
K26 74LS08
K27 74LS08
K28 74LS08
K29 74LS08
K30 74LS08
K31 74LS08
K32 74LS08
K33 74LS08
K34 74LS08
K35 74LS08
K36 74LS08
K37 74LS08
K38 74LS08
K39 74LS08
K40 74LS08
K41 74LS08
K42 74LS08
K43 74LS08
K44 74LS08
K45 74LS08
K46 74LS08
K47 74LS08
K48 74LS08
K49 74LS08
K50 74LS08
K51 74LS08
K52 74LS08
K53 74LS08
K54 74LS08
K55 74LS08
K56 74LS08
K57 74LS08
K58 74LS08
K59 74LS08
K60 74LS08
K61 74LS08
K62 74LS08
K63 74LS08
K64 74LS08
K65 74LS08
K66 74LS08
K67 74LS08
K68 74LS08
K69 74LS08
K70 74LS08
K71 74LS08
K72 74LS08
K73 74LS08
K74 74LS08
K75 74LS08
K76 74LS08
K77 74LS08
K78 74LS08
K79 74LS08
K80 74LS08
K81 74LS08
K82 74LS08
K83 74LS08
K84 74LS08
K85 74LS08
K86 74LS08
K87 74LS08
K88 74LS08
K89 74LS08
K90 74LS08
K91 74LS08
K92 74LS08
K93 74LS08
K94 74LS08
K95 74LS08
K96 74LS08
K97 74LS08
K98 74LS08
K99 74LS08
K100 74LS08
K101 74LS08
K102 74LS08
K103 74LS08
K104 74LS08
K105 74LS08
K106 74LS08
K107 74LS08
K108 74LS08
K109 74LS08
K110 74LS08
K111 74LS08
K112 74LS08
K113 74LS08
K114 74LS08
K115 74LS08
K116 74LS08
K117 74LS08
K118 74LS08
K119 74LS08
K120 74LS08
K121 74LS08
K122 74LS08
K123 74LS08
K124 74LS08
K125 74LS08
K126 74LS08
K127 74LS08
K128 74LS08
K129 74LS08
K130 74LS08
K131 74LS08
K132 74LS08
K133 74LS08
K134 74LS08
K135 74LS08
K136 74LS08
K137 74LS08
K138 74LS08
K139 74LS08
K140 74LS08
K141 74LS08
K142 74LS08
K143 74LS08
K144 74LS08
K145 74LS08
K146 74LS08
K147 74LS08
K148 74LS08
K149 74LS08
K150 74LS08
K151 74LS08
K152 74LS08
K153 74LS08
K154 74LS08
K155 74LS08
K156 74LS08
K157 74LS08
K158 74LS08
K159 74LS08
K160 74LS08
K161 74LS08
K162 74LS08
K163 74LS08
K164 74LS08
K165 74LS08
K166 74LS08
K167 74LS08
K168 74LS08
K169 74LS08
K170 74LS08
K171 74LS08
K172 74LS08
K173 74LS08
K174 74LS08
K175 74LS08
K176 74LS08
K177 74LS08
K178 74LS08
K179 74LS08
K180 74LS08
K181 74LS08
K182 74LS08
K183 74LS08
K184 74LS08
K185 74LS08
K186 74LS08
K187 74LS08
K188 74LS08
K189 74LS08
K190 74LS08
K191 74LS08
K192 74LS08
K193 74LS08
K194 74LS08
K195 74LS08
K196 74LS08
K197 74LS08
K198 74LS08
K199 74LS08
K200 74LS08
K201 74LS08
K202 74LS08
K203 74LS08
K204 74LS08
K205 74LS08
K206 74LS08
K207 74LS08
K208 74LS08
K209 74LS08
K210 74LS08
K211 74LS08
K212 74LS08
K213 74LS08
K214 74LS08
K215 74LS08
K216 74LS08
K217 74LS08
K218 74LS08
K219 74LS08
K220 74LS08
K221 74LS08
K222 74LS08
K223 74LS08
K224 74LS08
K225 74LS08
K226 74LS08
K227 74LS08
K228 74LS08
K229 74LS08
K230 74LS08
K231 74LS08
K232 74LS08
K233 74LS08
K234 74LS08
K235 74LS08
K236 74LS08
K237 74LS08
K238 74LS08
K239 74LS08
K240 74LS08
K241 74LS08
K242 74LS08
K243 74LS08
K244 74LS08
K245 74LS08
K246 74LS08
K247 74LS08
K248 74LS08
K249 74LS08
K250 74LS08
K251 74LS08
K252 74LS08
K253 74LS08
K254 74LS08
K255 74LS08
K256 74LS08
K257 74LS08
K258 74LS08
K259 74LS08
K260 74LS08
K261 74LS08
K262 74LS08
K263 74LS08
K264 74LS08
K265 74LS08
K266 74LS08
K267 74LS08
K268 74LS08
K269 74LS08
K270 74LS08
K271 74LS08
K272 74LS08
K273 74LS08
K274 74LS08
K275 74LS08
K276 74LS08
K277 74LS08
K278 74LS08
K279 74LS08
K280 74LS08
K281 74LS08
K282 74LS08
K283 74LS08
K284 74LS08
K285 74LS08
K286 74LS08
K287 74LS08
K288 74LS08
K289 74LS08
K290 74LS08
K291 74LS08
K292 74LS08
K293 74LS08
K294 74LS08
K295 74LS08
K296 74LS08
K297 74LS08
K298 74LS08
K299 74LS08
K300 74LS08
K301 74LS08
K302 74LS08
K303 74LS08
K304 74LS08
K305 74LS08
K306 74LS08
K307 74LS08
K308 74LS08
K309 74LS08
K310 74LS08
K311 74LS08
K312 74LS08
K313 74LS08
K314 74LS08
K315 74LS08
K316 74LS08
K317 74LS08
K318 74LS08
K319 74LS08
K320 74LS08
K321 74LS08
K322 74LS08
K323 74LS08
K324 74LS08
K325 74LS08
K326 74LS08
K327 74LS08
K328 74LS08
K329 74LS08
K330 74LS08
K331 74LS08
K332 74LS08
K333 74LS08
K334 74LS08
K335 74LS08
K336 74LS08
K337 74LS08
K338 74LS08
K339 74LS08
K340 74LS08
K341 74LS08
K342 74LS08
K343 74LS08
K344 74LS08
K345 74LS08
K346 74LS08
K347 74LS08
K348 74LS08
K349 74LS08
K350 74LS08
K351 74LS08
K352 74LS08
K353 74LS08
K354 74LS08
K355 74LS08
K356 74LS08
K357 74LS08
K358 74LS08
K359 74LS08
K360 74LS08
K361 74LS08
K362 74LS08
K363 74LS08
K364 74LS08
K365 74LS08
K366 74LS08
K367 74LS08
K368 74LS08
K369 74LS08
K370 74LS08
K371 74LS08
K372 74LS08
K373 74LS08
K374 74LS08
K375 74LS08
K376 74LS08
K377 74LS08
K378 74LS08
K379 74LS08
K380 74LS08
K381 74LS08
K382 74LS08
K383 74LS08
K384 74LS08
K385 74LS08
K386 74LS08
K387 74LS08
K388 74LS08
K389 74LS08
K390 74LS08
K391 74LS08
K392 74LS08
K393 74LS08
K394 74LS08
K395 74LS08
K396 74LS08
K397 74LS08
K398 74LS08
K399 74LS08
K400 74LS08
K401 74LS08
K402 74LS08
K403 74LS08
K404 74LS08
K405 74LS08
K406 74LS08
K407 74LS08
K408 74LS08
K409 74LS08
K410 74LS08
K411 74LS08
K412 74LS08
K413 74LS08
K414 74LS08
K415 74LS08
K416 74LS08
K417 74LS08
K418 74LS08
K419 74LS08
K420 74LS08
K421 74LS08
K422 74LS08
K423 74LS08
K424 74LS08
K425 74LS08
K426 74LS08
K427 74LS08
K428 74LS08
K429 74LS08
K430 74LS08
K431 74LS08
K432 74LS08
K433 74LS08
K434 74LS08
K435 74LS08
K436 74LS08
K437 74LS08
K438 74LS08
K439 74LS08
K440 74LS08
K441 74LS08
K442 74LS08
K443 74LS08
K444 74LS08
K445 74LS08
K446 74LS08
K447 74LS08
K448 74LS08
K449 74LS08
K450 74LS08
K451 74LS08
K452 74LS08
K453 74LS08
K454 74LS08
K455 74LS08
K456 74LS08
K457 74LS08
K458 74LS08
K459 74LS08
K460 74LS08
K461 74LS08
K462 74LS08
K463 74LS08
K464 74LS08
K465 74LS08
K466 74LS08
K467 74LS08
K468 74LS08
K469 74LS08
K470 74LS08
K471 74LS08
K472 74LS08
K473 74LS08
K474 74LS08
K475 74LS08
K476 74LS08
K477 74LS08
K478 74LS08
K479 74LS08
K480 74LS08
K481 74LS08
K482 74LS08
K483 74LS08
K484 74LS08
K485 74LS08
K486 74LS08
K487 74LS08
K488 74LS08
K489 74LS08
K490 74LS08
K491 74LS08
K492 74LS08
K493 74LS08
K494 74LS08
K495 74LS08
K496 74LS08
K497 74LS08
K498 74LS08
K499 74LS08
K500 74LS08
K501 74LS08
K502 74LS08
K503 74LS08
K504 74LS08
K505 74LS08
K506 74LS08
K507 74LS08
K508 74LS08
K509 74LS08
K510 74LS08
K511 74LS08
K512 74LS08
K513 74LS08
K514 74LS08
K515 74LS08
K516 74LS08
K517 74LS08
K518 74LS08
K519 74LS08
K520 74LS08
K521 74LS08
K522 74LS08
K523 74LS08
K524 74LS08
K525 74LS08
K526 74LS08
K527 74LS08
K528 74LS08
K529 74LS08
K530 74LS08
K531 74LS08
K532 74LS08
K533 74LS08
K534 74LS08
K535 74LS08
K536 74LS08
K537 74LS08
K538 74LS08
K539 74LS08
K540 74LS08
K541 74LS08
K542 74LS08
K543 74LS08
K544 74LS08
K545 74LS08
K546 74LS08
K547 74LS08
K548 74LS08
K549 74LS08
K550 74LS08
K551 74LS08
K552 74LS08
K553 74LS08
K554 74LS08
K555 74LS08
K556 74LS08
K557 74LS08
K558 74LS08
K559 74LS08
K560 74LS08
K561 74LS08
K562 74LS08
K563 74LS08
K564 74LS08
K565 74LS08
K566 74LS08
K567 74LS08
K568 74LS08
K569 74LS08
K570 74LS08
K571 74LS08
K572 74LS08
K573 74LS08
K574 74LS08
K575 74LS08
K576 74LS08
K577 74LS08
K578 74LS08
K579 74LS08
K580 74LS08
K581 74LS08
K582 74LS08
K583 74LS08
K584 74LS08
K585 74LS08
K586 74LS08
K587 74LS08
K588 74LS08
K589 74LS08
K590 74LS08
K591 74LS08
K592 74LS08
K593 74LS08
K594 74LS08
K595 74LS08
K596 74LS08
K597 74LS08
K598 74LS08
K599 74LS08
K600 74LS08
K601 74LS08
K602 74LS08
K603 74LS08
K604 74LS08
K605 74LS08
K606 74LS08
K607 74LS08
K608 74LS08
K609 74LS08
K610 74LS08
K611 74LS08
K612 74LS08
K613 74LS08
K614 74LS08
K615 74LS08
K616 74LS08
K617 74LS08
K618 74LS08
K619 74LS08
K620 74LS08
K621 74LS08
K622 74LS08
K623 74LS08
K624 74LS08
K625 74LS08
K626 74LS08
K627 74LS08
K628 74LS08
K629 74LS08
K630 74LS08
K631 74LS08
K632 74LS08
K633 74LS08
K634 74LS08
K635 74LS08
K636 74LS08
K637 74LS08
K638 74LS08
K639 74LS08
K640 74LS08
K641 74LS08
K642 74LS08
K643 74LS08
K644 74LS08
K645 74LS08
K646 74LS08
K647 74LS08
K648 74LS08
K649 74LS08
K650 74LS08
K651 74LS08
K652 74LS08
K653 74LS08
K654 74LS08
K655 74LS08
K656 74LS08
K657 74LS08
K658 74LS08
K659 74LS08
K660 74LS08
K661 74LS08
K662 74LS08
K663 74LS08
K664 74LS08
K665 74LS08
K666 74LS08
K667 74LS08
K668 74LS08
K669 74LS08
K670 74LS08
K671 74LS08
K672 74LS08
K673 74LS08
K674 74LS08
K675 74LS08
K676 74LS08
K677 74LS08
K678 74LS08
K679 74LS08
K680 74LS08
K681 74LS08
K682 74LS08
K683 74LS08
K684 74LS08
K685 74LS08
K686 74LS08
K687 74LS08
K688 74LS08
K689 74LS08
K690 74LS08
K691 74LS08
K692 74LS08
K693 74LS08
K694 74LS08
K695 74LS08
K696 74LS08
K697 74LS08
K698 74LS08
K699 74LS08
K700 74LS08
K701 74LS08
K702 74LS08
K703 74LS08
K704 74LS08
K705 74LS08
K706 74LS08
K707 74LS08
K708 74LS08
K709 74LS08
K710 74LS08
K711 74LS08
K712 74LS08
K713 74LS08
K714 74LS08
K715 74LS08
K716 74LS08
K717 74LS08
K718 74LS08
K719 74LS08
K720 74LS08
K721 74LS08
K722 74LS08
K723 74LS08
K724 74LS08
K725 74LS08
K726 74LS08
K727 74LS08
K728 74LS08
K729 74LS08
K730 74LS08
K731 74LS08
K732 74LS08
K733 74LS08
K734 74LS08
K735 74LS08
K736 74LS08
K737 74LS08
K738 74LS08
K739 74LS08
K740 74LS08
K741 74LS08
K742 74LS08
K743 74LS08
K744 74LS08
K745 74LS08
K746 74LS08
K747 74LS08
K748 74LS08
K749 74LS08
K750 74LS08
K751 74LS08
K752 74LS08
K753 74LS08
K754 74LS08
K755 74LS08
K756 74LS08
K757 74LS08
K758 74LS08
K759 74LS08
K760 74LS08
K761 74LS08
K762 74LS08
K763 74LS08
K764 74LS08
K765 74LS08
K766 74LS08
K767 74LS08
K768 74LS08
K769 74LS08
K770 74LS08
K771 74LS08
K772 74LS08
K773 74LS08
K774 74LS08
K775 74LS08
K776 74LS08
K777 74LS08
K778 74LS08
K779 74LS08
K780 74LS08
K781 74LS08
K782 74LS08
K783 74LS08
K784 74LS08
K785 74LS08
K786 74LS08
K787 74LS08
K788 74LS08
K789 74LS08
K790 74LS08
K791 74LS08
K792 74LS08
K793 74LS08
K794 74LS08
K795 74LS08
K796 74LS08
K797 74LS08
K798 74LS08
K799 74LS08
K800 74LS08
K801 74LS08
K802 74LS08
K803 74LS08
K804 74LS08
K805 74LS08
K806 74LS08
K807 74LS08
K808 74LS08
K809 74LS08
K810 74LS08
K811 74LS08
K812 74LS08
K813 74LS08
K814 74LS08
K815 74LS08
K816 74LS08
K817 74LS08
K818 74LS08
K819 74LS08
K820 74LS08
K821 74LS08
K822 74LS08
K823 74LS08
K824 74LS08
K825 74LS08
K826 74LS08
K827 74LS08
K828 74LS08
K829 74LS08
K830 74LS08
K831 74LS08
K832 74LS08
K833 74LS08
K834 74LS08
K835 74LS08
K836 74LS08
K837 74LS08
K838 74LS08
K839 74LS08
K840 74LS08
K841 74LS08
K842 74LS08
K843 74LS08
K844 74LS08
K845 74LS08
K846 74LS08
K847 74LS08
K848 74LS08
K849 74LS08
K850 74LS08
K851 74LS08
K852 74LS08
K853 74LS08
K854 74LS08
K855 74LS08
K856 74LS08
K857 74LS08
K858 74LS08
K859 74LS08
K860 74LS08
K861 74LS08
K862 74LS08
K863 74LS08
K864 74LS08
K865 74LS08
K866 74LS08
K867 74LS08
K868 74LS08
K869 74LS08
K870 74LS08
K871 74LS08
K872 74LS08
K873 74LS08
K874 74LS08
K875 74LS08
K876 74LS08
K877 74LS08
K878 74LS08
K879 74LS08
K880 74LS08
K881 74LS08
K882 74LS08
K883 74LS08
K884 74LS08
K885 74LS08
K886 74LS08
K887 74LS08
K888 74LS08
K889 74LS08
K890 74LS08
K891 74LS08
K892 74LS08
K893 74LS08
K894 74LS08
K895 74LS08
K896 74LS08
K897 74LS08
K898 74LS08
K899 74LS08
K900 74LS08
K901 74LS08
K902 74LS08
K903 74LS08
K904 74LS08
K905 74LS08
K906 74LS08
K907 74LS08
K908 74LS08
K909 74LS08
K910 74LS08
K911 74LS08
K912 74LS08
K913 74LS08
K914 74LS08
K915 74LS08
K916 74LS08
K917 74LS08
K918 74LS08
K919 74LS08
K920 74LS08
K921 74LS08
K922 74LS08
K923 74LS08
K924 74LS08
K925 74LS08
K926 74LS08
K927 74LS08
K928 74LS08
K929 74LS08
K930 74LS08
K931 74LS08
K932 74LS08
K933 74LS08
K934 74LS08
K935 74LS08
K936 74LS08
K937 74LS08
K938 74LS08
K939 74LS08
K940 74LS08
K941 74LS08
K942 74LS08
K943 74LS08
K944 74LS08
K945 74LS08
K946 74LS08
K947 74LS08
K948 74LS08
K949 74LS08
K950 74LS08
K951 74LS08
K952 74LS08
K953 74LS08
K954 74LS08
K955 74LS08
K956 74LS08
K957 74LS08
K958 74LS08
K959 74LS08
K960 74LS08
K961 74LS08
K962 74LS08
K963 74LS08
K964 74LS08
K965 74LS08
K966 74LS08
K967 74LS08
K968 74LS08
K969 74LS08
K970 74LS08
K971 74LS08
K972 74LS08
K973 74LS08
K974 74LS08
K975 74LS08
K976 74LS08
K977 74LS08
K978 74LS08
K979 74LS08
K980 74LS08
K981 74LS08
K982 74LS08
K983 74LS08
K984 74LS08
K985 74LS08
K986 74LS08
K987 74LS08
K988 74LS08
K989 74LS08
K990 74LS08
K991 74LS08
K992 74LS08
K993 74LS08
K994 74LS08
K995 74LS08
K996 74LS08
K997 74LS08
K998 74LS08
K999 74LS08
K1000 74LS08
by LUCA FUSINA AND CLAUDIO GRANUZZO
Schematic of MC68000/AIM 65 interface.

Figure 2

68000 On-Board Selections

Address (Hex)	Selection
8XX0	68000 lower 8 data bit (WRITE)
8XX1	68000 upper 8 data bit (WRITE)
8XX2	68000 output control signals (refer to Figure 4)
8XX3	68000 input control signals (refer to Figure 5)
8XX4	68000 lower 8 address bit
8XX5	68000 middle 8 address bit
8XX6	68000 upper 7 address bit
8XX7	68000 lower 8 data bit (READ)
8XX8	68000 upper 8 data bit (READ)
8XX9	$\overline{\text{HALT}}$
8XXA	N.C. (not connected)
8XXB	$\overline{\text{DTACK}}$

Figure 3

68000 Output Control Signals (IC22)

Bit No. on AIM Data Bus	Signal	
AD0	$\overline{\text{WRITE}}$	write
AD1	$\overline{\text{LDS}}$	lower data strobe
AD2	$\overline{\text{UDS}}$	upper data strobe
AD3	$\overline{\text{AS}}$	address select
AD4	$\overline{\text{BG}}$	bus grant
AD5	FC2	function code 2
AD6	FC1	function code 1
AD7	FC0	function code 0

Figure 4

68000 Input Control Signals (IC19)

Bit No. on AIM Data Bus	Signal	
AD0	$\overline{\text{BGACK}}$	bus grant ack
AD1	$\overline{\text{BERR}}$	bus error
AD2	N.C. (not connected)	
AD3	$\overline{\text{VPA}}$	valid peripheral address
AD4	$\overline{\text{BR}}$	bus request
AD5	IPL0	interrupt priority level 0
AD6	IPL1	interrupt priority level 1
AD7	IPL2	interrupt priority level 2

Figure 5

Meaning of the fields in the AIM listing of the 68000 program:

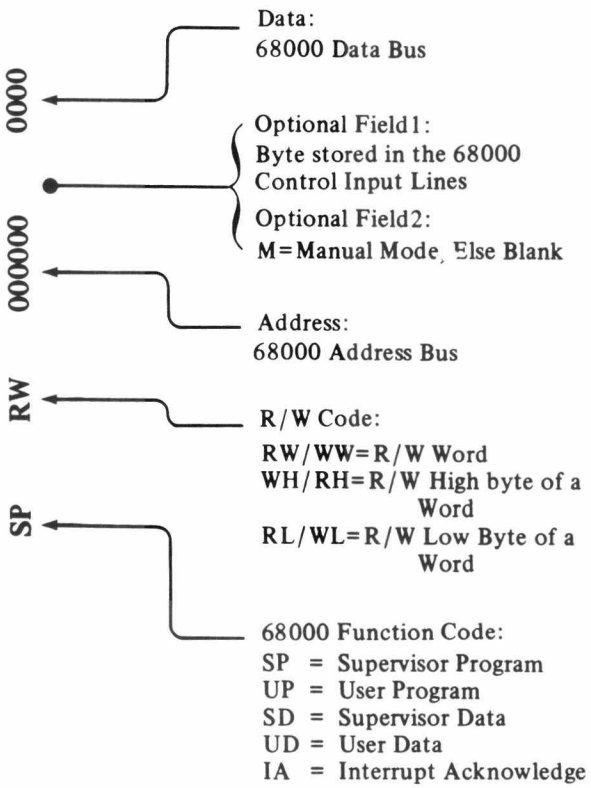


Figure 6

(Listing begins on page 36)

A Portable Screen-Oriented Editor

Edward K. Ream

Edward K. Ream, 1850 Summit Ave., Madison, WI 53705.

Are you tired of using a miserable line-oriented editor? Are you unwilling to pay several hundred dollars for a full-screen editor? Are you worried that your text editor will become obsolete if you change machines or operating systems? Are you frustrated that you can't make even simple changes to your editor? Are you unable to adapt your editor for more than one type of terminal? If so, here is an editor for you.

In the past five years I have used several different text editors on my computer system at home. My first editor was a typical character-oriented text editor. This editor did have some good points, but most of the time it forced me to type a lot more than I wanted to. Also, this editor was unsafe to use. Some mistakes that I could easily make caused the editor to lose many lines of text.

My second editor had more potential. It allowed me to do simple things simply. There were easy-to-use commands to move the cursor along a line, and to insert and delete characters on a line. The editor was also safe. Dangerous commands took more typing than harmless commands so editing was much less nerve wracking. I never destroyed more than a few moments of work with this editor.

This editor did have one major drawback: it didn't use the capabilities of the video screen I was using. As a rule, only one line of text was visible on the screen at a time and that line was always on the bottom of the screen. So while the editor had all the commands needed to edit files quickly, in practice the editor was difficult to use. I kept wanting to see the lines above and below the line where I was in order to make sure that I was making changes in the proper context.

I basically liked the editor so I modified it slightly so the screen would show several lines above the line being edited. This modification was actually a very small one; the editor still wasn't perfectly adapted to video displays (the cursor still couldn't move off the bottom line), but the editor was almost as easy to use as a full-screen editor.

Of course, I made other adjustments to that editor. I became fascinated with changing the editor's commands. I noticed a ripple effect; a change in the design of one command might lead me to modify five or six other commands. I spent many pleasant hours just thinking about how commands are interrelated.

After several months, I thought I had a pretty good editor. It was safe. It was predictable. Although complicated editing tasks sometimes took considerable work, simple editing jobs were very simple.

This editor had only one flaw, but that turned out to be fatal. It was written in assembly language. The editor worked fine until I got a new set of disks and a new operating system (CP/M). It soon became woefully apparent that almost 50% of the editor would have to be rewritten in order to transport it from the old operating system to the new. It looked as though I would be spending many weeks on this unpleasant project.

As luck would have it an alternative presented itself just at this time. *Dr. Dobb's Journal* published two articles by Ron Cain in which he presented a compiler for a language he called Small-C. (See *DDJ* #45 and #48, May and September, 1980.) Small-C is a subset of the full langauge called C, one of the best languages for writing systems programs like compilers and editors. Small-C would be perfect as a language in which to write a new editor. Just as importantly, the compiler for Small-C was written in Small-C itself. Although it may not be immediately obvious, that meant that the compiler formed the basis of a completely portable computer system. All programs written in Small-C could be moved from one machine to another just by changing a few routines in the compiler. The compiler was in the public domain, which meant I could use it or modify it however I wanted to. Clearly, the Small-C compiler was a great tool.

I obtained an excellent version of this compiler for the CP/M operating system from a company called The Code Works (TM). I recommend this compiler. It is well documented; it works reliably; all the source for the compiler is provided; and it's all in the public domain. At $17 it is one of the best bargains around.

After becoming acquainted with this compiler, I started work on what I hope will be the last version of this editor. Remember that the only thing wrong with my previous editor was that I couldn't move it easily to a new operating system. Thus, portability became a top priority. I wanted this editor to adapt easily to new operating systems or new display devices.

The Small-C editor shares many features with the old assembly language editor. I saw no reason to change a successful design. As a result, the new editor is as simple to use and reliable as the old editor was. However, the new editor has been completely rewritten. This has allowed me to improve many of the internal details of the editor's operation. I have also added full-screen-editing capability and a prompt line.

As far as I can tell the new editor is very portable. It has been used on widely differing terminals without making any programming changes. The editor comes with a configuration program which asks questions about your terminal. The configuration program then automatically produces two files that tailor the editor to your particular keyboard and display screen. I have not moved the editor to any other operating system besides CP/M, but I don't expect great problems in doing so. All code that depends on a particular operating system has been isolated to a few routines all located on one file. Moving to a new system requires only that these few routines be rewritten.

Basic Features of the Screen Editor

I hope by now you are eager to learn what this editor can do. Let's take a closer look at it. We will start by talking about some of the basic concepts and terms that are used throughout to describe the editor.

The screen editor is comprised of several *modes*. You can switch between modes, but the editor can't be in two modes at the same time. The editor behaves differently in each mode, so you use each mode to do a particular type of editing.

There are three modes: edit mode, command mode and insert mode. You use edit mode for making small changes in many different lines. Command mode is used for making larger (hence potentially more dangerous) changes to the whole file you are editing. You must be in command mode to do anything that will change files on the disk. You use insert mode for making a series of insertions into the text. First drafts are often entered in this mode.

You use *commands* to make things happen in each mode. Which commands you can use depends on what mode you

are in. Do not confuse the terms "commands" and "command mode." Every mode has a set of commands which you can use in that mode. Command mode has its own set of commands.

Special characters are one-letter commands. Special characters must be control keys, so you can use these commands in places where the editor might not otherwise expect a command.

The *prompt line* is the top line of the screen. It tells you things like what line you are editing and what mode you are in. If you get confused about what mode you are in, you can always glance at the prompt line and get your bearings.

Figure 1 shows what the prompt line looks like. The line field tells what line of the file you are currently editing. The column field tells what column on the screen the cursor is on. The column field mainly helps you align columns of text correctly. The next field tells you the name of the file you are currently editing. This field is set and used by several command-mode commands which will be discussed later on. The last field tells you what mode you are in.

Several edit-mode commands consist of two or more letters. These edit-mode commands are called *extended commands*. After you type the first letter of such a command, but before you type the last letter, the prompt line will indicate that you must type one more letter to finish the extended command. For example, after you type an "s" to enter the edit-mode search command the prompt line's mode field will contain "edit: search".

The *cursor* is a distinctive character on the screen. On most video terminals this marker is a character shown with reverse video. In command mode you always type commands on the bottom line so the cursor is always on the bottom line. In edit mode and insert mode the cursor is always on the current line.

The *current line* is the only line you can edit. In command mode the current line may not be shown on the screen, but in edit and insert modes the current line is shown and it contains the cursor. If you want to make a change to a line you must first move the cursor to that line.

The *buffer* is a part of the computer's storage which holds the file you are editing. In order to edit a disk file, you must first read the disk file into the buffer using the load command. As you edit the file, the changes you make are made only to the buffer, not to your disk file.

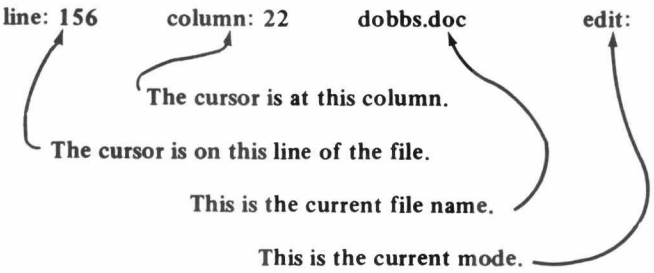


Figure 1: The Prompt Line
The prompt line tells where the cursor is and what the editor is doing.

Before you end your editing session you must copy the buffer back to a disk file using the save or resave commands. The editor keeps track of whether you have done this; it won't let you forget to save your work.

The *window* is a part of the screen which shows a portion of the buffer. In edit and insert modes the window fills all of the screen except the prompt line. You can think of the file as a scroll of parchment which unwinds under the window as you move the cursor. The window is not shown in command mode.

Now let's look more closely at the editor's commands. We'll start with special characters, which are commands that may be used in either insert or edit modes. Then we will look at other commands which are unique to a particular mode.

Special Characters

Special characters act the same in all modes. Special characters must be control characters so that the editor may always distinguish special characters from regular text.

There are 14 special characters; all 14 may be used in either edit mode or insert mode. Only a few may be used in command mode. Figure 2 shows how to switch between modes using three of these keys. Figure 3 summarizes what all these special keys do.

You can use the configuration program to choose which keys on your keyboard to use for each special character. Thus, I won't be able to be specific about what key on your keyboard you will actually hit in order to type, for instance, the down key. I'll often refer to special characters as special keys; for example, I'll call the down special character the down key.

The *left and right keys* just move the cursor. They do not cause changes to lines. These keys are always "anchored" to the current line; you can't move off the current line by using these keys. If you hit the right key while the cursor is at the right margin, nothing will happen. Similarly, nothing will happen if you hit the left key while the cursor is at the left margin.

The *up and down keys* move the cursor up and down on the screen. You can't move the cursor above line 1 of the file, nor can you move the cursor below the last line of the file. Both these keys also switch the editor to edit mode.

The *insert up key* inserts a blank line above the current line while the *insert down key* inserts a blank line below the current line. Both these keys also switch the editor to insert mode.

The *delete character key* deletes the character to the left of the cursor. This key is anchored to the current line; nothing is deleted if the cursor is at the left margin. The *delete line key* deletes the entire line on which the cursor rests.

The *undo key* always undoes whatever editing you have done on the current line since the last time the cursor came to the current line.

The *insert key* switches the editor to insert mode. Similarly, the *command key* switches the editor to command mode and the *edit key* switches the editor to edit mode.

The *split key* splits the current line into two pieces. Everything to the left of the cursor stays right where it is. All other characters are moved from the current line. Both these keys also switch the editor to insert mode.

The *join key* is the opposite of the split key; it combines two lines into one line. The join key appends the current line to the line above it, then deletes the lower line.

Edit Mode

Edit-mode commands are normal letters; you don't need to use special keys. This speeds typing. You can, however, use the special keys in edit mode if you want to.

Besides the special characters you can use the following one-letter edit-mode commands: space, b, c, d, e, g, i, k, s, u, and x. These letters stand for begin, command, down, end, go, insert, kill, search, up, and eXchange. Note that all these commands may be typed either in upper case or in lower case. Letters which are neither edit-mode commands nor special keys are simply ignored. Figure 4 is a brief summary of these commands. The following paragraphs explain what these commands do in greater detail.

The *space bar* moves the cursor right one column. Nothing happens if the cursor is already on the rightmost column of the screen.

The *b command* puts the cursor at the beginning (left-hand margin) of the line.

The *c command* switches the editor to command mode.

The *d command* causes the cursor to move down rapidly. The cursor keeps moving until it reaches the last line of the file or until you type any key.

The *e command* moves the cursor to the right end of the line.

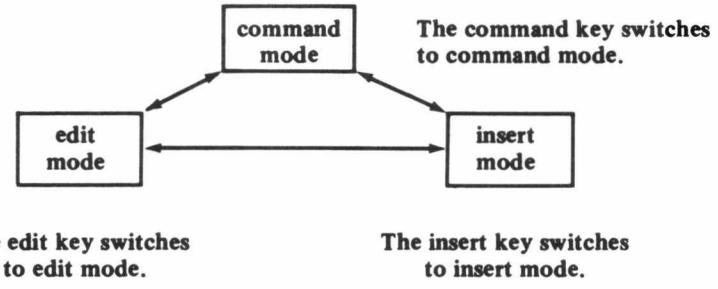


Figure 2: The Modes of the Editor
The edit, insert and command special characters switch the editor from mode to mode.

Special Character	Function
up key:	move cursor up 1 row & enter edit mode
down key:	move cursor down 1 row & enter edit mode
*left key:	move cursor left 1 character
right key:	move cursor right 1 character
insert up key:	insert line above current line & enter insert mode
insert down key:	insert line below current line & enter insert mode
*delete character key:	delete character to left of the cursor
*delete line key:	delete the current line
*undo key:	undo all changes to the current line
*insert key:	enter insert mode
*edit key:	enter edit mode
command key:	enter command mode
split key:	split the current line into two lines
join key:	combine two lines into one line

Figure 3: Special Characters
Special characters are one-character commands that are valid in edit mode and insert mode. Special characters which may be used in command mode are indicated by an asterisk (*).

The *i command* switches the editor to insert mode.

The *g command* moves the cursor to another line. The g command is an extended command; after you type the g, the cursor will move to the prompt line. The prompt line will show "edit: goto:" Now type a line number followed by a carriage return. The cursor will move to the line whose number you typed. If you do not type a valid number the g command does nothing. Leading blanks or minus signs are not allowed. The cursor will move to the last line of the buffer if the number you type is larger than the number of lines in the buffer.

The *k command* is a two-letter command. The second letter you type (the first character after the k) is a search character. The k command deletes from the cursor up to but not including the next occurence of the search character to the right of the cursor. Everything from the cursor to the end of the line is deleted if the search character does not appear to the right of the cursor on the current line. After you hit the k and before you hit the search character the prompt line will show "edit: kill". If you wish to cancel the k command you can hit any control character. The k command will be stopped and no deletion will be made.

The *s command* is another two-letter command. The second character you type (the first character after the s) is a search character. The s command moves the cursor to the next occurence of the search character which appears to the right of the cursor. The cursor moves to the end of the current line if the search character does not occur to the right of the cursor. After you hit the s and before you hit the search character the prompt line will show "edit: search".

The *u command* moves the cursor up rapidly. The cursor keeps moving until it reaches the first line of the file or until you hit any key.

The *x command* is another two-letter command. The second character you type replaces the character under the cursor. The prompt line will show "edit: eXchange" until you hit the second character. If you hit a control character no change is made and the x command is cancelled.

Insert Mode

Use insert mode to insert many lines of text at once. In insert mode anything

you type (except special keys) will be inserted into the text to the left of the cursor. You may use special keys in insert mode exactly as in edit mode.

Command Mode

You use command mode to do potentially dangerous things like making multiple changes to the buffer or like updating your files. For this reason, all command-mode commands require you to type several characters with a carriage return at the end.

The editor starts off in command mode after you invoke it from your operating system.

You may use either the edit key or the insert key to exit from command mode. You can also use the g command-mode command or the find command-mode command to leave command mode. When the editor leaves command mode the screen is redrawn to show the current line.

The cursor is on the bottom line of the screen whenever the editor is in command mode. This line is called the command line. As you type commands, what you type will be shown on this line. Use either the delete character key or the right key to delete the last character you typed. Use the undo key to delete the entire command line.

Several command-mode commands take arguments. There are three types of arguments: range arguments, file names and line numbers. In the description below, range arguments will be denoted by <line range>, file name arguments will be denoted by <filename> and line numbers will be denoted by <n>.

Range arguments denote the range of lines in the buffer for which a command will take effect. For example,

```
list 25 50
```

means list all lines from line 25 to line 50 inclusive,

```
list
```

means list all the lines of the buffer, and

```
list 300
```

means list line 300 and all following lines. Range arguments, if present, must be nonnegative integers. If two numbers are entered, the first must be no larger than the second. 0 is equivalent to 1. A number larger than the largest line number is equivalent to the largest line number.

Filename arguments are the name of a file to be used in a command. For example,

```
load myfile.doc
```

causes the file myfile.doc to replace whatever is in the buffer. Note that no quotation marks are used around filename arguments. Filename arguments optionally may be preceded by disk drive names. For example,

```
load b:myfile.doc
```

loads the buffer from the file myfile.doc which must be found on drive b.

The editor protects you from destroying the buffer unintentionally. If a command would cause the buffer to be erased and you have not already saved the buffer on a disk file the editor will ask:

```
buffer not saved. proceed ?
```

Command	Mnemonic	Function
space	(over)	move cursor right 1 character
b	(Begin)	move cursor to start of line
c	(Command)	enter command mode
d	(Down)	scroll screen down
e	(End)	move cursor to end of line
g<n>	(Go)	go to line number <n>
i	(Insert)	enter insert mode
k<char>	(Kill)	delete from cursor up to <char>
s<char>	(Search)	move cursor right to H char K
u	(Up)	scroll screen up
x<char>	(eXchange)	replace cursor with <char>

Figure 4: Edit Mode Commands
Edit mode commands are one letter long. They may be typed in either upper or lower case.

If you type y or Y, the command will be done and the buffer will be erased. If you type anything else no change will be made to the buffer and the message "cancelled" will be printed on the screen. Note, however, that the delete command does *not* protect you in this way. Use the delete command with caution.

You may interrupt or cancel the change, find, list and search commands. If you type any character *except* a blank the command will be cancelled immediately. If you type a blank the command will be temporarily suspended. Type another blank to continue, or any other character to cancel the command.

The following paragraphs discuss all the command-mode commands. Figure 5 gives a short summary of how to invoke these commands and how to use them.

append <filename>

The append command inserts the named file into the buffer after the current line. In other words, the position of the cursor affects where the file will be inserted. This command adds to the buffer; it does not delete what is already there.

change <line range>

The change command searches all lines in <line range> for an instance of a search mask. If the search mask is found it is replaced by the change mask. After you type change <line range> the command will ask:

```
search mask ?
```

At this point type the pattern you are searching for, followed by a carriage return. Then you will be asked:

```
change mask ?
```

Now you type the pattern you want the search mask to be replaced by. For example,

```
change 100 300
search mask ? abc
change mask ? xyz
```

will change abc to xyz in all lines from 100 to 300. Thus, the line:

```
know your abc's
```

would be changed into:

```
know your xyz's
```

Only the first instance of abc on each line would be changed. The line:

```
know your abc's. The abc's are
    very important.
```

would be changed to:

know your xyz's. The abc's are
very important.

Question marks may be used in both the search mask and the change mask. A question mark in the search mask matches any character at all. The search mask:

12?34

will match any string of five character which starts with 12 and ends with 34. A question mark in the change mask matches the letter that matched the corresponding question mark in the search mask. For example:

change 99 109
search mask ? trailing?
change mask ? ?leading

would change the line:

trailingA

into the line:

Aleading

The question mark in the search mask matched the "A", so the question mark in the change mask became an "A". You can use as many question marks as you like, but there must be at least as many question marks in the search mask as there are in the change mask. As another example:

change
search mask ? wh??e
change mask ? fath??

would change the line:

where is here.

to the line:

father is here.

If the first character of the search mask is an up arrow (↑), the search mask is anchored to the start of the line. In other words, the search mask will only match patterns which start at column 0. For example, the search mask:

↑abc

will only match lines that contain abc in columns 0, 1 and 2. Anchoring the search mask dramatically increases the speed of the change command.

An up arrow which is not the first character of the search mask is treated as a regular up arrow. An up arrow in the change mask never has any special significance.

clear

The clear command erases the entire buffer. If you have not saved the buffer you will be asked whether you want to proceed.

delete <line range>

The delete command erases all lines in the line range. Care must be used with this command because no check is made to see whether the buffer has been saved.

dos

The dos command exits the editor and returns to the operating system. If the buffer has not been saved, this command makes sure that you really wanted to exit without saving the buffer.

find

The find command searches for the next occurrence of a search mask. If the search mask is found, the editor switches to edit mode and the cursor is set to the start of the pattern. If the search mask is not found the editor remains in command mode.

You specify the search mask exactly as in the change command.

g <n>

The g command exits to edit mode. Line <n> is drawn on the top line of the window and becomes the current line. Typing g without the <n> is the same as typing the edit key.

list <line range>

The list command lists all lines in the line range to the current list device, i.e., the printer. Lines are formatted just as they are on the screen, but the length of the print line, not the width of the screen, determines where lines will be truncated if they are too long. Listing may be stopped at any time by typing any key.

load <filename>

The load command replaces whatever is in the buffer by the contents of the named file. If the buffer has not been saved on disk when the load command is given, the editor makes sure you really wanted to erase the previous contents of the buffer. If the named file exists, then <filename> becomes "the current file" and <filename> is shown on the prompt line. This file becomes the file used in the save and resave commands. The current file name does not change if the named file does not exist.

After the file has been completely read from the disk into the buffer the screen is redrawn to show the first several lines of the file.

name <filename>

The name command sets the name of the file that will be used in the save and resave commands. This command is not often used because the load command is the usual way to set this file's name. The name command does not cause any action immediately; it simply affects which file the save and resave commands will use.

resave

The resave command saves the entire buffer in the current file. The name of the current file is set by either the load or name commands and always appears on the prompt line. Note that this command does *not* take any arguments. All arguments to this command are ignored.

The resave command requires that the current file already exist; if the file does not exist a warning is printed and nothing happens.

save

The save command works just like the resave command except that the file must *not* exist for the command to work. If the file does exist a warning is printed and nothing happens. This protection ensures that files cannot be overwritten inadvertently.

search <line range>

The search command prints on the screen all lines in <line range> which contain an instance of the search mask. As soon as the search command is typed the editor will ask:

search mask ?

Now type the search mask. Just as in the change command, a question mark matches any character and a leading up arrow anchors the search mask to the start of the line.

tabs <n>

The tabs command controls how lines are listed on the screen and on the printer. Specifically, the tabs command sets tab stops every <n> columns. <n> must be an integer. The command tabs 0 is equivalent to the command tabs 1.

Bringing Up & Changing the Editor

Figure 6 shows what equipment you need to have in order to use the editor without making *any* changes to the program. The code in the listings should work on any 8080, Z80, or 8085 machine that runs the CP/M operating system. You will also need quite a bit of memory. The editor itself uses about 23K bytes and the file you are editing must fit in the memory that is left. You also need a display screen that can be made to directly address the cursor. By that I mean that there must be some way to move the cursor wherever you like in one step. Every video board and terminal that I am aware of has this capability.

If you don't have access to the CP/M operating system then you will certainly have to make at least a few changes to the editor in order to get it up and running. But don't despair quite yet; you should only have to change the programs in the file ed8.c. Basically, these routines mask the idiosyncrasies of your operating system from the rest of the editor. If you find that you must change a routine not on this file in order to get the editor running on some other system, I'd like to hear about it. But I don't think you will.

Those of you with the CP/M 1.4 operating system will have to make only a very minor change to the routines syscstat(), syscin() and syscout() on file ed8.c. The listings contain the CP/M 1.4 versions of these routines commented out. Just comment out the CP/M 2.2 versions and remove the comments from around the CP/M 1.4 versions.

Those of you with displays that *only* work in block mode are going to have to work pretty hard to use this editor. (If you don't know what a block mode terminal is, rejoice. You don't have one of the beasts.) A somewhat uneducated guess is that you may have to rewrite all the routines in file ed6.ccc and many of the routines in file ed2.c. Frankly, I don't think it is worth the trouble. If somebody out there does get the editor working with a block-mode terminal I'd like to hear about *that*, too.

The files ed1.ccc and ed6.ccc contain code which is produced by the configuration program. These files tailor the editor to a particular display device. If for some reason you do not want to use the configuration program, you will have to create your own versions of these files.

One of the things that I learned while doing this project is that it is possible to change drastically how the editor looks to the user without really changing the guts the editor at all. From a programming point of view, the choice of which commands to include and which to exclude is pretty much arbitrary. Barry Dobyns' article about the MINCE editor in the April 1981 issue of *Dr. Dobb's Journal* is a good source of ideas about what you might want to add to this editor.

Everyone will have a favorite command or two that has been left out of this editor. I miss being able to move blocks of text from one area of the file to another. Another improvement might be to have a new line generated automatically in insert mode if the cursor reaches the end of the line. This is essentially an "auto-split" feature. These changes should be easy to make if you are so inclined.

The present version of this program limits editing to those files that can fit entirely into memory. This is not a fundamental problem with the editor. In fact, the first thing I designed about the editor was a scheme to edit arbitrarily long files. Essentially, what one has to do is to rewrite about half of the buffer module (file ed10.c). If there is enough interest, I may write another article telling how to do this.

Command	Arguments	Function
append	<filename>	insert file into buffer at cursor
change	<line range>	make changes to lines in <line range>
clear		erase the buffer
delete	<line range>	deletes lines in <line range>
dos		exit from editor to operating system
find		search for a pattern; enter edit mode
g	<n>	enter edit mode and go to line <n>
list	<line range>	print lines in <line range> to printer
load	<filename>	replace buffer with file
name	<filename>	set name of current file
save		save buffer to a new file
search	<line range>	print all lines that match a pattern
resave		save buffer to existing file
tabs	<n>	set tab stops at every <n> columns

Figure 5: Command Mode Commands
All command mode commands end with a carriage return. They may be typed in either upper or lower case.

How to Get This Editor

This editor is in the public domain. That means you can use it in any way you wish. You can move the editor to other machines, copy it however you like, and make whatever changes you think would make it better. In fact, I strongly encourage you to do these things.

If you have the CP/M operating system, you can get this editor and several other programs from me. They come on two 8-inch, soft-sectored, single-density, IBM-compatible diskettes. On these diskettes are the following:

1. Source code for the editor.
2. Source code for the configuration program.
3. Source code for the Small-C compiler as distributed by The Code Works (TM).
4. Source code for a text formatter adapted from the FORMAT program in the book, *Software Tools*, by Kernighan and Plauger, published by Addison-Wesley.
5. Complete documentation.

The cost for all the above programs is $50. I am making the cost somewhat higher than the cost of just copying the disks because I would prefer to send out just a few copies. It's fine with me if you get your copy from somebody who already has one.

I cannot offer to provide extensive support for these programs. That might quickly take all of my time. I will, however, pay one dollar to anyone who finds a bug in this program. Sorry, that's the best I can do.

If you use these programs and find them useful I hope you will add something of your own to them. I think this is a good editor, but I'm sure some of you will think of improvements. I'd like to

read about your ideas in *Dr. Dobb's Journal.*

Technical Details

Now that we have seen what the editor does, you might be interested in the technical details of how the editor does what it does. You can skip this section if these details are of no use to you. But please, if you do intend to make changes to this editor, make sure you understand the way the editor is constructed. If you don't, you are going to do violence to the design behind the editor.

The screen editor is organized into separate modules, also known as abstract data types. If you understand how and why the program is organized into these modules then you understand the basics of the program. I followed three rules religiously while writing this program. These rules tell how modules are designed. If you modify this program I want you to follow them too. Here they are:

Rule 1:

No program may use any data contained in another module. That is, all access to data contained in a module must be via the "access routines" of that module.

Rule 2:

No access routine may reveal to the outside world the internal representation of data inside the access routine's own module. In other words, the access routines of a module only indicate what the module does, not how the module does it.

Rule 3:

Every significant (i.e., relatively complex) data structure resides in one and only one module.

Each module, then, is responsible for hiding the details about some important data structure. To be more precise, each module hides how the module's internal data is represented. The idea is that the organization of the data may be changed inside a module without the programs that use the module knowing that anything has changed.

Modules not only contain data; just as importantly they also contain access routines by which procedures outside the module may use or change the data. The access routines are to be considered part of the module's data structure. I will sometimes speak of the "associated operations" of a module or data structure. By that I will mean the externally visible access routines of the module.

Since modules (or data structures) have associated operations, the concept of data structure widens to include dynamic devices such as disk drives, video screens, and operating systems! Indeed, I use modules to shield the rest of the program from the details of such devices.

Rule 1 says that programs outside a module can only use or change the data inside the module by calling the "access routines" of the module. No messing with other module's data is allowed.

Rule 2 says that these access routines are functions of the data. That is, the values returned and the effects produced by the access routines are independent of how the data of the module is represented. Rule 2 eliminates data dependencies between modules.

Rule 2 doesn't quite say enough. Not only must access routines hide their modules representation of data, access routines must provide a functional view of the module to the outside world. In other words, there must not be complex dependencies between access functions in the same modules. (By definition there are not dependencies between access functions of different modules.) Access functions can be linked in subtle ways. For example, it might be that the access functions of a module must be called in only certain orders. Such dependencies should be avoided if at all possible.

Rule 3 says that all important data structures are protected from meddling from outside sources, or conversely, no program needs to know (or can know) about details of data structures outside it's own module.

A corollary of these rules is that there are no important global data structures. Indeed, there is seldom, if ever, a need for any global variables at all. Variables may be shared by more than one routine, but all such routines will reside in the same module.

Not every routine needs to be a member of a module. Those routines that are not a member of any module know the details of none of the important data structures of the program.

There is a certain optimal complexity for the data structure enclosed by a module. If the structure is too complex then too many routines (all of them inside the module!) know details of the data structure. On the other hand, if a

(Continued on page 34)

Computer: 8080, 8085, or Z80 (4 mhz or faster preferred)

Operating system: CP/M

Main Memory: at least 32K (editor uses 23K)

Display: video screen with direct cursor addressing

Figure 6: What You Need to Use the Editor
You will have to make changes to the editor if you don't have the kind of system listed above.

Name of Module	File Name
window module	ed4.c
buffer module	ed10.c
line format module	ed5.c
prompt line module	ed7.c
terminal output module	ed6.ccc
operating system module	ed8.c
command module	ed2.c

Figure 7: Modules
The editor is made up of 7 modules. Each module controls an important data structure. Each module is on a separate file; modules do not share data.

Small Shell

Part 2 of a North* VOS

My last article (*DDJ* #61) described Small-VM, the first component of a virtual operating system (VOS) for North Star machines. It adds file management, memory management, and program control features to a standard, unmodified North Star DOS. It also constitutes a run-time library for Small-C and programs compiled by Small-C. This article presents the second component of VOS – a command processor dubbed Small-Shell.

My objectives in designing the shell were:

1. ease of use (user friendly)
2. application worthiness (user proof)
3. conceptual simplicity (clean design with no frills)
4. small size (this version is 2K bytes)
5. shell penetration (access to the underlying DOS)
6. compatibility with all North Star software
7. independence (works alone or with Small-VM)
8. procedure files (with parameter substitution, operator prompting, command override, and abnormal procedure termination)
9. no DOS irritants (command line too short, DEL does not rub out, superfluous keyword GO, danger of accidental IN command)
10. save and simple reentry of serially reusable programs.

Small-Shell is written in assembly language for Allen Ashley's PDS macro-assembler. Using assembly language violates the spirit of the VOS concept which emphasizes portability, and portability is a major goal of the VOS project; but I feel it is more important to keep the size of memory-resident system software down on a microcomputer with limited memory. Writing the shell in C also would have made it dependent on Small-VM; as it stands it may be used either with or without that system.

A complete listing of Small-Shell accompanies this article. Small-Shell is copyrighted and is published here for private non-commercial use only.

Basic Concept of the Shell

The shell exists as a self-contained

J.E. Hendrix

J. E. Hendrix, Rt. 1, Box 74-B-1, Oxford, MS 38655.

Listing

```
;
;*****************************************************************
;* Small/Shell: Version 1.0
;*              A command processor for North Star DOS
;*
;* Copyright    1981  J. E. Hendrix
;*              All rights reserved
;*
;* Written for PDS MAKRO assembler
;*****************************************************************
;
;----- DOS references
;
DOSBASE: EQU     2000H
;
DOS:     EQU     DOSBASE+28H ;;DOS WARM START
CIN:     EQU     DOSBASE+10H ;;CHAR INPUT
COUT:    EQU     DOSBASE+0DH ;;CHAR OUTPUT
HDERR:   EQU     DOSBASE+19H ;;HARD DISK ERROR BRANCH
DOSERR:  EQU     DOSBASE+2CH ;;DLOOK FILENAME ERROR BRANCH
DLOOK:   EQU     DOSBASE+1CH ;;DIRECTORY LOOKUP
DCOM:    EQU     DOSBASE+22H ;;RD, WRT, VERIFY DISK
CMDBF:   EQU     DOSBASE+31H ;;DOS CMD BUFFER POINTER
;
;----- non-standard DOS references
;
DOSCCP:  EQU     DOSBASE+0B1FH;;DOS CMD PROC (LXI H,dosbf)
DOSPR:   EQU     DOSBASE+0B19H;;DOS PROMPT (CALL nnnn)
;
;----- misc. assembly constants
;
BLKSZ:   EQU     256         ;;DISK BLOCK SIZE
DDEN:    EQU     2           ;;DRIVE DENSITY
DCOMR:   EQU     1           ;;DCOM READ COMMAND
PAD:     EQU     0           ;;SMALL/VM BUFFER PAD CHAR
CTLC:    EQU     03H         ;;^C
CTLE:    EQU     05H         ;;^E
CTLS:    EQU     13H         ;;^S
COMT:    EQU     ';'         ;;COMMENT FLAG
DOSCMD:  EQU     '+'         ;;TARGET DOS FOR CMD
JUMP:    EQU     '.'         ;;JUMP TO LOADED PROGRAM
NULPARM: EQU     '.'         ;;NULL PARAMETER FLAG
NOTE:    EQU     '_'         ;;OPERATOR NOTE FLAG IN PROCEDURE FILE
PROM:    EQU     '%'         ;;PROMPT CHAR
SUBMARK: EQU     '$'         ;;MARKS SYMBOLIC PARAMETER IN PROCEDURE FILE
SYSFILE: EQU     '*'         ;;IDENTIFIES A SYSTEM FILE
CONSOLE: EQU     0           ;;NORTH STAR CONSOLE DEVICE
EXEC:    EQU     1           ;;TYPE OF EXECUTABLE FIEL
PROC:    EQU     11          ;;TYPE OF PROCEDURE FILE
CLSZ:    EQU     62          ;;SHELL COMMAND LINE SIZE
DEFDRV:  EQU     1           ;;DEFAULT DRIVE FOR PROG AND PROC LOOKUP
;
;----- initializing
;
         LXI     H,BOSTACK   ;;SET SP TO BOTTOM OF STACK
         SPHL
         LHLD    CMDBF       ;;SAVE/ALTER DOS CMD BUFFER VECTOR
         SHLD    CMDBFV
         LXI     H,CMDLINE
         SHLD    CMDBF
         INX     H           ;;ALTER DOS CMD PROC TO SEE SHELL CMD LINE
         SHLD    DOSCCP+1
         LHLD    DOSPR+1     ;;SAVE/ALTER DOS PROMPT CALL
         SHLD    DOSPRV
         LXI     H,RETURN
         SHLD    DOSPR+1
         LHLD    HDERR+1     ;;SAVE/ALTER DOS HARD ERROR VECTOR
         SHLD    HDERRV
         LXI     H,HARDERR
         SHLD    HDERR+1
         LHLD    DOSERR+1    ;;SAVE FILENAME ERROR VECTOR
         SHLD    DOSERRV
         LHLD    DOS+1       ;;SAVE/ALTER DOS WARM START VECTOR
         SHLD    DOSV
         LXI     H,RETURN
         SHLD    DOS+1
         SHLD    LADDR       ;;INIT LOAD ADDR
```

(Continued on next page)

Small Shell (Listing continued, text begins on page 27)

North Star program which may be invoked with the GO command. It is in control when other programs are not running. Shell commands either invoke a program or a series of commands from a procedure file. When a running program returns to DOS, an altered jump-table address diverts control back to the shell. When the shell regains control from a program, it checks to see if control came through Small-VM; if so, then it checks for a non-zero return code which would terminate a procedure if one were in progress. The command to invoke a procedure looks like a program execution command; the shell learns the difference by examining the file type. Regular DOS commands may be pushed through the shell for DOS processing. One may exit the shell by entering a control-E, but this feature, and the DOS commands as well, may be disabled for non-privileged applications. Once out of the shell, it may be reentered by issuing a JP to its load address. The shell absolutely refuses to honor the IN command; one must exit to DOS to format diskettes.

Command Syntax

Regular (non-DOS) shell commands have the form "filename arg1 arg2 arg3 . . ." where filename is a standard North Star file name with an optional drive number (e.g., prog,2). The arguments are optional depending upon the requirements of the individual program or procedure. If the named file is of type I, it is loaded and control is passed to its load address with HL pointing to the first command line character following filename. If the file is of type II, it is taken for a procedure file; the individual lines within it are processed sequentially. Nesting of procedures is not supported. If the file is not found or if it is an illegal type of file, the command is rejected by displaying "?" followed by the full command (after parameter substitution). A null line or any line with leading spaces is ignored.

In place of filename, one may enter a period. This instructs the shell to jump to the last program loaded (after setting LH appropriately); this feature saves diskette I/O and typing while performing repetitive operations.

The Shell, like UNIX (UNIX is a trademark of Bell Labs), prompts with a percent sign (%). Anything may be used, but it should differ from the symbol used by DOS ("+" for double-density sys-

```
78 ;
79 ;----- do next shell command
80 ;
81 DOCMD:   ENTRY
82          LXI     H,CRERR         ;;ALTER DOSERR HERE SINCE BASIC CORRUPTS IT
83          SHLD    DOSERR+1
84          LDA     SKIPSW          ;;SKIP THIS CMD?
85          ANA     A
86          MVI     A,0
87          STA     SKIPSW
88          JNZ     NEXTCMD         ;;YES
89          LDA     CMDLINE         ;;FIRST BYTE OF CMD
90          CPI     0DH             ;;CR?
91          JZ      NEXTCMD         ;;YES, IGNORE CMD
92          CPI     ' '             ;;LEADING SPACE WOULD TELL DLOOK TO
93          JZ      NEXTCMD         ;;  FIND 1ST AVAILABLE SECTOR
94 CHKSYS:  CPI     0               ;;SYS FILE? <<<<ALTERED AFTER BOOT>>>>
95          JZ      CMDERR
96 ;
97 ; RETURN TO DOS?
98 ;
99          CPI     CTLE            ;;^E?
100         JNZ     DOCMD2          ;;NO
101         LDA     DOSPRIV         ;;YES, IS DOS PRIVILEGE SET?
102         ANA     A
103         JZ      NEXTCMD         ;;NO -- TOO BAD
104         LHLD    DOSV            ;;RESTORE DOS WARM START VECTOR
105         SHLD    DOS+1
106         LHLD    CMDBFV          ;;RESTORE DOS CMD BUFFER VECTOR
107         SHLD    CMDBF
108         SHLD    DOSCCP+1        ;;RESTORE DOS CCP
109         LHLD    DOSPRV          ;;RESTORE DOS PROMPT
110         SHLD    DOSPR+1
111         LHLD    HDERRV          ;;RESTORE HARD ERR VECTOR
112         SHLD    HDERR+1
113         LHLD    DOSERRV         ;;RESTORE FILENAME ERROR VECTOR
114         SHLD    DOSERR+1
115         MVI     A,JUMP          ;;PREPARE FOR REENTRY TO SHELL
116         STA     CMDLINE
117         JMP     DOS             ;;<---
118 ;
119 ; JUMP TO A PREVIOUSLY LOADED PROGRAM?
120 ;
121 DOCMD2:  CPI     JUMP
122         JZ      GOPROG
123 ;
124 ;----- DOS command?
125 ;
126         CPI     DOSCMD
127         JNZ     LOOK            ;;NO
128         LDA     CMDLINE+1       ;; "IN" CMD?
129         CPI     'I'
130         JZ      CMDERR          ;;YES, TOO DANGEROUS (MUST EXIT TO DOS)
131         CPI     'i'             ;;YES, "in"
132         JZ      CMDERR
133         CPI     0DH             ;; "+" ALONE?
134         JNZ     DOCMD9          ;;NO
135         LDA     PROCSW          ;;YES, IN A PROC?
136         ANA     A
137         JZ      CMDERR          ;;NO
138         LDA     DOSPRIV         ;;YES, TOGGLE DOS PRIV SW
139         CMA
140         STA     DOSPRIV
141         JMP     NEXTCMD
142 ;
143 DOCMD9:  LDA     DOSPRIV         ;;DOS PRIV SET?
144         ANA     A
145         JZ      CMDERR          ;;NO
146         STA     CMDLINE-1       ;;TELL SMALL/VM IT'S A DOS COMMAND
147         JMP     DOSCCP
148 ;
149 ;----- lookup named file
150 ;
151 LOOK:    LXI     H,CMDLINE       ;;HL -> FILENAME
152         MVI     A,DEFDRV        ;;A = DEFAULT DRIVE
153         CALL    DLOOK           ;;LOOK FOR IT <---
154         JC      CMDERR          ;;NOT FOUND
155         STA     DRV             ;;SAVE ACTUAL DRIVE
156         XCHG                    ;;FDE+8 IN DOS
157         LXI     H,DA
158         XCHG
159         MVI     C,7
160         CALL    BMOVE           ;;SAVE FDE FIELDS
161 ;
162         LDA     DRV             ;;COMBINE DENSITY BIT AND DRIVE #
163         MOV     C,A
164         LDA     FTY
165         MOV     B,A
```

```
166             ANI     80H             ;;DENSITY BIT
167             ORA     C
168             STA     DRV
169 ;
170             MOV     A,B             ;;EXECUTABLE FILE?
171             ANI     7FH
172             CPI     EXEC
173             JZ      LOAD            ;;YES
174             CPI     PROC            ;;PROCEDURE FILE?
175             JZ      DOPROC          ;;YES
176             JMP     CMDERR          ;;NO -- TOO BAD
177 ;
178 ;----- load executable file
179 ;
180 LOAD:   LHLD    TDI             ;;UPDATE LOAD ADDR
181             SHLD    LADDR
182             LDA     DRV
183             MOV     C,A             ;;C = DEN, DRV
184             MVI     B,DCOMR         ;;B = DCOM READ COMMAND
185             LDA     FSZ             ;;A = FILE SIZE IN SECTORS (255 MAX)
186             LHLD    LADDR
187             XCHG                    ;;DE = LOAD ADDRESS
188             LHLD    DA              ;;HL = DISK ADDRESS
189             CALL    DCOM            ;;LOAD IT <---
190             JC      SHELLERR        ;;TROUBLE
191 ;
192 ;----- jump to loaded program
193 ;
194 GOPROG: LHLD    LADDR
195             SHLD    GO+1            ;;ALTER JMP ADDR
196             LXI     H,CMDLINE       ;;SET HL PER NORTH STAR PRACTICE
197 GOLOOP: INX     H
198             MOV     A,M
199             CPI     21H             ;;1ST CHAR <= SPACE?
200             JP      GOLOOP          ;;NO
201             XRA     A               ;;TELL SMALL/VM IT'S A SHELL COMMAND
202             STA     CMDLINE-1
203 GO:     ENTRY
204             JMP     SHELLERR        ;;<<<< THIS ADDR IS ALTERED >>>>
205 ;         .
206 ;         .
207 ;         .
208 RETURN: ENTRY
209             XCHG                    ;;DE = RETURN CODE IF FROM SMALL/VM
210             LXI     H,BOSTACK       ;;SET SP
211             SPHL
212             XRI     0FFH            ;;RETURNED VIA SMALL/VM?
213             JNZ     NEXTCMD         ;;NO
214             ORA     E               ;;YES, IS LSB OF RETURN CODE ZERO?
215             JNZ     UNPROC          ;;NO, ABORT PROC CMD IF IN PROGRESS
216 ;
217 ;----- where is next command to be found?
218 ;
219 NEXTCMD: LDA     PROCSW
220             ANA     A
221             JNZ     PFETCH
222 ;
223 ;----- console input
224 ;
225 PROMPT: MVI     B,PROM          ;;PROMPT CHAR
226             CALL    OUTCHAR
227             LXI     H,CMDLINE       ;;HL = DEST PTR
228             MVI     C,CLSZ          ;;C = CHAR COUNTER
229 CINLOOP: MVI     A,CONSOLE
230             CALL    CIN             ;;FETCH NEXT CHAR <---
231             CPI     08H             ;;BS?
232             JZ      RUBOUT
233             CPI     7FH             ;;DEL?
234             JZ      RUBOUT
235             CPI     CTLC            ;;^C?
236             JNZ     STOWIT          ;;NO
237             MVI     A,0DH           ;;YES
238             STA     CMDLINE         ;;IGNORE CMD
239 STOWIT: MOV     M,A
240             INX     H
241             DCR     C
242             JZ      LINERR          ;;NO MORE SPACE
243             CPI     20H             ;;CTL CHAR?
244             JP      ECHO            ;;NO, CONTINUE
245             MVI     A,0DH           ;;FORCE CR
246             MOV     M,A
247 ECHO:   MOV     B,A             ;;ECHO CHAR
248             CALL    OUTCHAR         ;;RETURNS CHAR IN "A"
249             CPI     0AH             ;;OUTCHAR GENERATES AUTO LF AFTER CR
250             MVI     A,0             ;;CLEAR SKIPSW
251             STA     SKIPSW
252             JZ      DOCMD
253             JMP     CINLOOP
254 ;
255 RUBOUT: MOV     A,C
256             CPI     CLSZ
257             JZ      CINLOOP         ;;CMDLINE IS EMPTY
```

(Continued on next page)

tems). One should know whether he is conversing with DOS or the shell.

DOS commands are entered with a leading plus sign; using the DOS prompt character seems to have mnemonic value. These commands are passed through the shell to DOS for processing.

Procedure Files

A procedure file is a simple ASCII file with each line terminated by a CR or a CR/LF sequence. The file is terminated by (1) a hex FF, (2) a zero byte, or (3) the physical end of file space. The last line is ignored if it is not properly terminated.

Any valid console command is legal in a procedure. In addition, the following commands have special meaning only in a procedure:

:<comment>

Any line beginning with a colon is taken for a comment and is ignored by the shell.

_<note>

Any line beginning with a single underscore is displayed as a note to the operator.

__<prompt>

Any line beginning with two underscores is also displayed. In this case, however, the message "waiting . . ." is then sent to the operator and the shell waits for a response. This serves two purposes. It permits the operator to change diskettes, ready the printer, etc., and it lets the operator decide whether to (1) continue normally, (2) skip the next command, or (3) cancel the procedure.

A carriage return response continues the procedure. A control-S response skips the next command. Comments do not count as commands, but notes and null commands do. A command may be made unskippable by preceding it with a note or a blank line. A control-C response cancels the procedure. Other responses cause the shell to redisplay the waiting message.

+

A line consisting of a single plus sign (+) toggles the DOS-access privilege. Since the privilege is initially turned on, placing this command in the boot procedure automatically denies the operator DOS access. The privilege may be reinstated by processing another toggle command from a procedure file. This scheme is not foolproof, but it is adequate and it is not

```
258           INR     C
259           DCX     H
260           PUSH    H
261           LXI     H,RUBMSG
262           CALL    OUTSTR
263           POP     H
264           JMP     CINLOOP
265 ;
266 ;----- procedure input
267 ;
268 PFETCH:  MVI     A,CLSZ          ;;BYTES LEFT IN CMDLINE
269           STA     CMDBYTES
270           LXI     H,CMDLINE       ;;NEXT CMDLINE DEST ADDR
271           SHLD    CMDADDR
272           XRA     A               ;;CLEAR SUBSTITUTION FLAG
273           STA     SUBFLAG
274 ;
275 ; IS A NEW SECTOR NEEDED?
276 ;
277 PLOOP:   LHLD    MCNT            ;;CHARS LEFT IN BUFFER
278           MOV     A,H
279           ORA     L
280           JNZ     BUFMOVE         ;;NO, MOVE NEXT CHAR
281 ;
282 ; ANY MORE SECTORS IN THE FILE?
283 ;
284           LHLD    MFSZ
285           MOV     A,H
286           ORA     L
287           JZ      UNPROC          ;;NO, END OF FILE
288 ;
289 ; READ NEXT SECTOR
290 ;
291           LDA     MDRV
292           MOV     C,A             ;;C = DEN, DRV
293           MVI     B,DCOMR         ;;B = DCOM READ COMMAND
294           MVI     A,1             ;;A = # SECTORS TO READ
295           LXI     D,MBUF          ;;DE = BUFFER ADDR
296           LHLD    MDA             ;;HL = DISK ADDR
297           CALL    DCOM            ;;READ 1 SECTOR <---
298           JC      SHELLERR        ;;TROUBLE
299           LXI     H,BLKSZ*DDEN    ;;INIT MCNT
300           SHLD    MCNT
301           LXI     H,MBUF          ;;INIT NEXT BUFFER ADDR
302           SHLD    MADDR
303           LHLD    MDA             ;;BUMP PROC DISK ADDR
304           INX     H
305           SHLD    MDA
306           LHLD    MFSZ            ;;DECR PROC FILE SIZE (SECTORS LEFT)
307           DCX     H
308           SHLD    MFSZ
309 ;
310 ;----- move next byte to cmdline from procedure buffer
311 ;
312 BUFMOVE: LHLD    MADDR
313           MOV     A,M
314           INX     H
315           SHLD    MADDR
316           LHLD    MCNT            ;;DECR MCNT
317           DCX     H
318           SHLD    MCNT
319           PUSH    PSW             ;;PREV CHAR A SUBSTITUTION MARKER?
320           LDA     SUBFLAG
321           ANA     A
322           JNZ     PARMSUB         ;;YES
323           POP     PSW             ;;NO
324           CPI     SUBMARK         ;;THIS CHAR A SUBSTITUTION MARKER?
325           JNZ     BUFMOV2         ;;NO
326           STA     SUBFLAG         ;;YES, SET FLAG
327           JMP     PLOOP           ;;AND CONTINUE
328 ;
329 BUFMOV2:  CPI     0FFH            ;;PDS EOF FLAG?
330           JZ      UNPROC          ;;YES, EOF
331           CPI     PAD             ;;SMALL/VM PAD CHAR?
332           JZ      UNPROC          ;;YES, EOF
333 BUFMOV3:  CPI     0AH             ;;LF?
334           JZ      PLOOP           ;;YES, IGNORE IT
335 BUFPUT:   LHLD    CMDADDR         ;;NEXT CMDLINE ADDR
336           MOV     M,A
337           CPI     0DH             ;;CR?
338           JZ      PEOL            ;;YES, CMD IS FETCHED
339           INX     H
340           SHLD    CMDADDR
341           LDA     CMDBYTES        ;;CMDLINE BYTES LEFT
342           DCR     A
343           STA     CMDBYTES
344           ANA     A               ;;SPACE LEFT?
```

(Continued on page 32)

difficult for a knowledgeable person to override (edit the command out of the boot file and reboot or use a procedure on a different diskette to toggle the privilege back on). One cannot create a procedure file containing this command since the privilege is required to set its file type to II.

The shell, like UNIX, employs a positional scheme for parameter substitution in procedures. Up to 9 parameters may follow filename when a procedure is invoked. Within the procedure, these are referenced symbolically as $1, $2, $3, . . . $9. The symbol $0 refers to filename. These references may appear anywhere on any procedure line regardless of the type of line. Operator notes, for instance, may vary according to the parameters given when the procedure was invoked.

Since the parameter substitution logic takes no account of syntax, symbolic references may be used in any context; they may form parts of fields, complete fields, or complete lines. A dollar sign followed by any non-numeric character is dropped, but the following character is retained; thus, $$ yields $.

If a referenced parameter is not given with the invoking command, the reference becomes a null string. Trailing parameters, when invoking a procedure, may be omitted by entering nothing after the last parameter given. Other parameters may be skipped by entering a period in their places. Thus, the command "abc . . . xx" invokes procedure abc replacing all references to $0 with abc, and $3 with xx; all other references are squeezed out.

Two switches may be included with commands to invoke procedures. They may appear anywhere following the procedure name, but are not seen as parameters. The switch "-v" (meaning "view") causes the shell to display each command after parameter substitution but before being processed. Comments are not displayed. The switch "-n" (meaning "negate") causes processing of the commands to be skipped. The command "abc parm1 parm2 -v" will show the progress of procedure abc. The command "abc parm1 parm2 -v -n" only displays the procedure text after parameter substitution; this is handy for debugging new procedures.

Booting a System With the Shell

The shell's procedure processing fea-

ture is a handy device for booting up a system of separate self-contained modules. It is much neater and easier to extend the DOS that way than to modify its internals. That approach also results in a more flexible system. My first use of procedure files was to boot a system consisting of DOS, the shell, a video driver, and Small-VM. Setting the autostart byte of DOS to zero tells it to automatically process whatever command is in its buffer at boot time. Having "GO *SHELL" in the DOS command buffer, then, automatically invokes the shell.

The shell always executes the command "*BOOT" when it is first invoked. A procedure file by that name resides on the diskette; it contains:

```
*VIDEO
*VM
_VOS Development Diskette
```

Presto! The video driver and the VM interface are initiated and the diskette identifies itself. One could easily give the operator prerogative over which components to boot by placing prompt commands in the boot procedure.

The leading asterisk in the file names *BOOT, *SHELL, *VIDEO, and *VM identifies them to the shell as system files which are illegal to reference after boot time. This mechanism prevents the user from corrupting the system by invoking system modules or the boot procedure a second time.

A Word of Warning

Small-Shell makes two non-standard references to DOS. DOSCCP locates the point where DOS loads its command line buffer address in preparation for processing a command. The shell alters the address at DOSCCP+1 to trick DOS into using the shell buffer instead of its own. It jumps to DOSCCP when a DOS command is ready to process. The second reference, DOSPR, locates a point traversed within the DOS just before it prompts for a new command. By altering DOSPR+1, the shell diverts control back to itself after a DOS command is finished.

These references will likely need changing if the shell is to be used with any version of DOS other than release 5.2DQ. The context of these reference points is:

```
          MVI B,28H
            ("+" prompt symbol)
DOSPR:    CALL <output>
          CALL <input>
DOSCCP:   LXI H,<buffer>
            (address stored at CMDBF)
          LXI B,<whatever>
          MOV A,M
```

To facilitate communication between Small-VM and Small-Shell when transferring control to and from program, Small-VM has been modified. The following partial listing highlights these changes with a vertical bar to the left of each modified line: *(See below)*

This publication is available in microform.

University Microfilms International

300 North Zeeb Road
Dept. P.R.
Ann Arbor, Mi. 48106
U.S.A.

30-32 Mortimer Street
Dept. P.R.
London W1N 7RA
England


```
; DECR2:   CALL    GTSOB       ;;RET ONLY IF SOB < PTR
           DCX     H
           MOV     A,M
             .
             .
             .
  GTSOB:   XCHG
           LHLD    SOB
           XCHG                ;;DE=SOB, HL=PTR
           PUSH    H
           CALL    UCMP
           POP     H
           RC                  ;;RET IF SOB<PTR
           POP     B           ;;WAST RET ADDR
;          DCX     H           ;;IS THIS SMALL-SHELL TALKING?
;          MOV     A,M
;          ANA     A
;          JNZ     SKPA        ;;NO, IGNORE "GO" OR "JP"
           INX     H
           PUSH    H           ;;NEXT ARGV PTR
           DCX     H
           CALL    ARGS
             .
             .
             .
  CCEXIT:  ENTRY
;          POP     H           ;;WAST RET ADDR
;          POP     H           ;;ERROR CODE?
;          SHLD    TEMP
;          MOV     A,H
;          ORA     L
           JZ      EXITO       ;;NO
;          MOV     B,L
  ABORT:   XRA     A
;          STA     TEMP+1
           CALL    COUT
;          STA     TEMP
           MVI     B,3FH       ;;"?"
           XRA     A
           CALL    COUT
             .
             .
             .
           LXI     H,AMAX+FMAX+2;;LAST FILE FD
           CALL    UCMP        ;;DONE?
;          LHLD    TEMP        ;;SET UP RET CODE FOR SMALL-SHELL
;          MVI     A,0FFH      ;;TELL SMALL-SHELL A SMALL-VM PROG ENDED
           JZ      DOS         ;;YEP
             .
             .
             .
```

(Text continued on following page)

```
345             JZ      LINERR          ;;NO, ABORT PROC
346             JMP     PLOOP           ;;YES, GO FOR NEXT BYTE
347 ;
348 ;----- parameter substitution (PSW on top of stack)
349 ;
350 PARMSUB:   XRA     A               ;;CLEAR SUB FLAG
351             STA     SUBFLAG
352             POP     PSW             ;;FETCH CHAR FOLLOWING SUBST MARKER
353             PUSH    PSW             ;;AND SAVE IT AGAIN
354             SUI     30H             ;;REDUCE ASCII BYTE TO INTEGER
355             JM      NOSUB           ;;LT ZERO
356             CPI     10
357             JP      NOSUB           ;;GT 9
358             MOV     C,A             ;;PPTR OFFSET
359             POP     PSW             ;;WAST TOP OF STACK
360             LXI     H,PPTRS         ;;1ST PPTR ADDR
361 SUBLOOP:   MOV     E,M             ;;DE = NEXT PPTR
362             INX     H
363             MOV     D,M
364             INX     H
365             MOV     A,D
366             ORA     E
367             JZ      NULLPARM
368             DCR     C
369             JP      SUBLOOP
370 ;
371 ; PPTR FOUND
372 ;
373 SUBLOOP2:XCHG                   ;;HL -> NEXT PARM CHAR
374             MOV     A,M             ;;FETCH IT
375             CPI     21H             ;;END OF PARM (LE SPACE)?
376             JM      PLOOP           ;;YES, CONTINUE
377             INX     H               ;;BUMP PARM BYTE ADDR
378             XCHG                    ;;DE -> FOLLOWING PARM BYTE
379             LHLD    CMDADDR         ;;STORE BYTE IN CMDLINE
380             MOV     M,A
381             INX     H               ;;BUMP BYTE ADDR
382             SHLD    CMDADDR
383             LDA     CMDBYTES        ;;DECR BYTES LEFT IN CMDLINE
384             DCR     A
385             STA     CMDBYTES
386             ANA     A               ;;SPACE LEFT?
387             JZ      LINERR          ;;NO, ABORT PROC
388             JMP     SUBLOOP2
389 ;
390 ; NON-NUMERIC CHAR FOLLOWS SUBSTITUTION MARKER -- TREAT NORMAL
391 ;
392 NOSUB:    POP     PSW             ;;RESTORE CHAR
393             JMP     BUFMOV2         ;;AND CONTINUE
394 ;
395 ; NO PARAMETER GIVEN FOR SUBSTITUTION -- SUB NULL STRING
396 ;
397 NULLPARM:MVI     A,0AH           ;;FORCE BYTE TO BE SKIPPED
398             JMP     BUFMOV3
399 ;
400 ;----- end of procedure line
401 ;
402 PEOL:     LDA     CMDLINE
403             CPI     COMT            ;;COMMENT?
404             JZ      PFETCH          ;;YES, IGNORE IT
405             CPI     NOTE            ;;OPERATOR NOTE?
406             JZ      VIEW            ;;YES, DISPLAY THE LINE
407             LDA     VSW             ;;VIEW SWITCH SET?
408             ANA     A
409             JZ      CKNSW           ;;NO
410 VIEW:     LXI     H,CMDLINE       ;;YES
411             CALL    OUTSTR
412 ;
413 CKNSW:    LDA     NSW             ;;NO-EXECUTE SWITCH SET?
414             ANA     A
415             JNZ     PFETCH          ;;YES, GO FOR NEXT COMMAND LINE
416 ;
417             LDA     CMDLINE         ;;OPERATOR NOTE?
418             CPI     NOTE
419             JNZ     DOCMD           ;;NO, PROCESS THE COMMAND LINE
420             LDA     CMDLINE+1       ;;OPERATOR RESPONSE?
421             CPI     NOTE
422             JNZ     PFETCH          ;;NO
423 WAIT:     LXI     H,WAITMSG
424             CALL    OUTSTR
425             XRA     A               ;;WAIT FOR A RESPONSE
426             CALL    CIN             ;; <---
427             PUSH    PSW             ;;ECHO CR,LF
428             MVI     B,0DH
429             CALL    OUTCHAR
430             POP     PSW
431             CPI     0DH             ;;CR?
```

Future Plans

For the future I hope to write an assembler, a linkage editor, and some utility progrms in C. The linkage editor will operate disk-to-disk and will assume a default load address; it will automatically set its output file to the correct type. That will eliminate the need for routine use of the SF and TY commands. Utility programs are needed for directory listings, multiple file copying, renaming and deleting. That should eliminate the need for routine use of the LI and DE commands, making it unnecessary to access DOS under normal conditions.

The chief drawback to building on North Star DOS is its use of contiguous files. Once created, they cannot be dynamically resized. Small-VM partially overcomes this inconvenience by permitting the last file created to be dynamically expanded or contracted. It also dynamically creates files. I hope soon to explore a simple scheme for supporting multiple file extents. It would make all files expandable and contractable and make all unused space available without compacting the diskette. This can be done *without* (1) changing the simple DOS diskette format, (2) losing full compatibility with standard North Star files, or (3) decreasing available diskette space. The penalty in memory usage may make this feature unattractive though.

Many features could be added to the shell, but the ones presented here should meet most needs. One worthy addition might be the ability to run simultaneous foreground and background programs. If you have any ideas or suggestions, I would like to hear from you. **DDJ**

```
432             JZ      PFETCH          ;;YES, CONTINUE
433             CPI     CTLC            ;;^C?
434             JZ      UNPROC          ;;YES, ABORT PROC
435             CPI     CTLS            ;;^S?
436             JNZ     WAIT            ;;NO, KEEP WAITING
437             CMA                     ;;YES, SET SKIP SWITCH
438             STA     SKIPSW
439             JMP     PFETCH          ;;FETCH NEXT PROC COMMAND
440 ;
441 ;----- error routines
442 ;
443 CMDERR:     LXI     H,CMDLINE
444             JMP     ERROUT
445 LINERR:     MVI     B,0DH
446             CALL    OUTCHAR
447             LXI     H,LINEMSG
448             JMP     ERROUT
449 PARMERR:    LXI     H,PARMSG
450             JMP     ERROUT
451 SHELLERR:   LXI     H,SHELLMSG
452             JMP     ERROUT
453 HARDERR:    LXI     H,HARDMSG
454             JMP     ERROUT
455 CRERR:      LXI     H,CRMSG
456             JMP     ERROUT2
457 NESTERR:    LXI     H,NESTMSG
458 ;
459 ERROUT:     PUSH    H
460             LXI     H,QUESTMSG
461             CALL    OUTSTR
462             POP     H
463 ERROUT2:    CALL    OUTSTR
464 ;
465 ;----- terminate procedure processing
466 ;
467 UNPROC:     XRA     A
468             STA     PROCSW
469             MVI     A,SYSFILE       ;;REJECT FUTURE SYS FILE REFS
470             STA     CHKSYS+1
471             JMP     RETURN          ;;RESETS SP & GOES TO PROMPT
472 ;
473 ;----- initiate procedure processing
474 ;
475 DOPROC:     LDA     PROCSW          ;;ALREADY IN A PROC?
476             ANA     A
477             JNZ     NESTERR         ;;YES, BUT CAN'T NEST THEM
478             CMA                     ;;NO, SET PROC SWITCH
479             STA     PROCSW
480             LXI     H,MDRV          ;;SET UP PROC FDE FIELDS
481             XCHG
482             LXI     H,DRV
483             MVI     C,5
484             CALL    BMOVE
485             LXI     H,0             ;;FORCE FIRST DISK READ
486             SHLD    MCNT
487 ;
488 ;----- parse procedure parameters
489 ;
490             XRA     A
491             STA     VSW             ;;CLEAR VIEW SWITCH
492             STA     NSW             ;;CLEAR NO-EXECUTE SWITCH
493             LXI     H,PPTRS         ;;INIT PPADDR
494             SHLD    PPADDR
495             LXI     H,PBUF          ;;INIT DEST ADDR (DE)
496             XCHG
497             LXI     H,CMDLINE       ;;INIT SOURCE ADDR (HL)
498             MVI     B,10            ;;MAX PARMS ALLOWED
499             MVI     C,0             ;;C = WORD SWITCH (ZERO BETWEEN WORDS)
500 ;
501 PARSLOOP:   MOV     A,M             ;;LOAD NEXT BYTE
502             INX     H               ;;BUMP SOURCE ADDR
503             CPI     '-'             ;;CMD LINE SWITCH?
504             JNZ     PARS1           ;;NO
505             MOV     A,C             ;;BEGINNING OF A WORD?
506             ANA     A
507             JNZ     PARSOB          ;;NO, CONTINUE
508             INX     H
509             MOV     A,M             ;;TEST 3RD BYTE
510             DCX     H
511             CPI     21H             ;;SPACE OR CTL CHAR?
512             JP      PARSOB          ;;NO, CONTINUE
513             MOV     A,M             ;;TEST 2ND BYTE
514             CPI     'V'             ;;-V?
515             JNZ     PARSOA          ;;NO
516             STA     VSW             ;;YES, SET VIEW SWITCH
517 PARSO:      INX     H               ;;AND DROP SWITCH FROM THE COMMAND
518             JMP     PARSLOOP
519 ;
520 PARSOA:     CPI     'N'             ;;-N?
521 PARSOB:     MVI     A,'-'           ;;RESTORE CHAR
522             JNZ     PARS1           ;;NO, CONTINUE
```

(Continued on next page)

Clinic *(Continued from page 7)*

```
BRA     .+5             ; skip around the
                          calling arguments
FCB     operation       ; 0 for wait, -1 for
                          signal
FDB     semaphore       ; address of the
                          semaphore byte
```

An interrupt routine can return through the *signal* routine to record an event and to possibly restart a task of higher priority than the current one. The call would look like this:

```
LDX     #semaphore      ; point index regis-
                          ter at the sema-
                          phore byte
JMP     Signal          ; transfer to the
                          signal routine
```

"Note that the kernel can return control to the proper task by simply executing an interrupt return instruction, RTI."

Laurino's letter ended with the question, "Can you find a fast algorithm to replace the iterative loop in the dispatcher that doesn't involve a tree of bit tests and branches? It is a kind of 'identify the next element in the set' function."

Help!

Despite many favorable comments, the initial heavy mail for the Clinic has dropped off to a trickle. Your input is essential; without it this column is going to wither away. The range of material that can be used is very wide, and the effort to send something in is quite small. The Clinic needs: operating system idiocies, clever algorithms, sticky problems, brain teasers, unusual facts explained, striking fallacies exposed . . . in short, almost anything in software or hardware that interests you or gives you pause will also interest or surprise the other readers. Send 'em in!

[1] Donald E. Knuth, *The Art of Computer Programming* (Volume 1, *Fundamental Algorithms*), Second Edition. Addison-Wesley, 1973, page 190.

[2] *Ibid*, page 226.

(Continued from page 24)

structure is too simple, then wrapping a module around it increases the overall conceptual overhead of the program. Note that a single variable may often be suitable as the data structure of a module if the associated operations of the module are sufficiently complex. For example, the program counter in an assembler might make an ideal abstract data type.

The editor program is composed of 7 modules. Each module resides on a separate file. Thus, in regular C all global variables in these 7 modules could be declared to be static. This would prevent any module from accessing the variables of other modules. Figure 7 shows what each of these modules does. ■■J

(Listing begins on page 42.)

Author Ream's Editor is available on disk from the BDS C User's Group (see ad on page 17).

Small Shell (Listing continued, text begins on page 27)

```
523             STA     NSW             ;;YES, SET NO-EXECUTE SWITCH
524             JMP     PARSO
525 ;
526 PARS1:      XCHG
527             MOV     M,A
528             INX     H
529             XCHG
530             CPI     ODH             ;;END OF LINE?
531             JZ      PARSEND         ;;YES
532             CPI     21H             ;;SPACE OR CTL CHAR?
533             MOV     A,C
534             JP      PARS2           ;;NO
535             ANA     A               ;;YES, IS WORD SWITCH CLEAR?
536             JZ      PARSLOOP        ;;YES, CONTINUE
537             MVI     C,0             ;;NO, CLEAR IT AND CONTINUE
538             JMP     PARSLOOP
539 ;

540 PARS2:      ANA     A               ;;IS WORD SWITCH SET?
541             JNZ     PARSLOOP        ;;YES, CONTINUE
542             MVI     C,1             ;;NO, SET IT
543             DCR     B               ;;WILL NEXT PARM FIT PPTRS ARRAY?
544             JM      PARMERR         ;;NO
545 ;
546 ; LOAD NEXT PARM PTR
547 ;

548             PUSH    H               ;;SOURCE
549             PUSH    D               ;;DEST
550             MOV     A,M             ;;2ND SOURCE CHAR <= SPACE?
551             CPI     21H
552             XCHG
553             DCX     H

554             JP      PARS3           ;;NO, CAN'T BE NULL PARAMETER
555             MOV     A,M             ;;1ST DEST CHAR A NULL PARAMETER?
556             CPI     NULPARM
557             JNZ     PARS3           ;;NO
558             MVI     A,' '           ;;YES, FORCE PARM DELIMITER
559             MOV     M,A

560 PARS3:      XCHG
561             LHLD    PPADDR
562             MOV     M,E
563             INX     H
564             MOV     M,D
565             INX     H
566             SHLD    PPADDR
567             POP     D               ;;DEST
568             POP     H               ;;SOURCE
569             JMP     PARSLOOP
570 ;
571 ;----- terminate parsing
572 ;

573 PARSEND:    XRA     A               ;;CAP PPTRS WITH A NULL WORD
574             LHLD    PPADDR
575             MOV     M,A
576             INX     H
577             MOV     M,A
578             JMP     PFETCH          ;;GO PROCESS THE PROC
579 ;
580 ;----- block move routine

581 ;       HL -> source,  DE -> dest, C = length
582 BMOVE:     MOV     A,M
583             INX     H
584             XCHG
585             MOV     M,A
586             INX     H
587             XCHG
588             DCR     C
589             JNZ     BMOVE
590             RET

591 ;
592 ;----- output a char to the console
593 ;       B = char, returns char in "A", auto LF
594 OUTCHAR:    MVI     A,CONSOLE
595             CALL    COUT            ;; <---
596             CPI     ODH             ;;CR?
597             RNZ                     ;;NO
598             MVI     B,0AH           ;;YES, GENERATE LF
```

```
599             MVI     A,CONSOLE
600             CALL    COUT            ;; <---
601             RET
602 ;
603 ;----- output a string to the console
604 ;       HL -> string, ends after CR or before NULL
605 OUTSTR:   MOV     B,M
606             MOV     A,B
607             ANA     A
608             RZ
609             INX     H
610             CALL    OUTCHAR
611             CPI     OAH             ;;LF
612             JNZ     OUTSTR
613             RET
614 ;
615 ;----- misc variables
616 ;

617 CMDADDR:  DW      0               ;;CUR CMDLINE ADDR
618 CMDBYTES: DB      0               ;;CMDLINE BYTES LEFT
619 DOSPRV:   DW      0               ;;DOS PROMPT VECTOR
620 CMDBFV:   DW      0               ;;DOS CMD BUFFER VECTOR
621 DOSV:     DW      0               ;;DOS WARM START VECTOR
622 HDERRV:   DW      0               ;;DOS HARD ERROR VECTOR
623 DOSERRV:  DW      0               ;;DOS FILENAME ERROR VECTOR
624 PROCSW:   DB      0               ;;PROC SWITCH
625 SKIPSW:   DB      0               ;;SKIP NEXT PROC CMD
626 VSW:      DB      0               ;;VIEW SWITCH
627 NSW:      DB      0               ;;NO-EXECUTE SWITCH
628 SUBFLAG:  DB      0               ;;PARM SUBST FLAG
629 DOSPRIV:  DB      OFFH            ;;DOS ACCESS PRIVILEGE FLAG

630 ;
631 ;----- shell command line
632 ;
633             DB      0               ;;TELL'S SMALL/VM WHETHER DOS OR SHELL COMMAND
634 CMDLINE:  DB      '*BOOT',ODH     ;;INITIAL COMMAND
635             DS      CLSZ-6
636 ;
637 ;----- program FDE & control fields
638 ;


639 DRV:      DB      0               ;;DEN (BIT 7), DRIVE
640 DA:       DW      0               ;;DISK ADDR
641 FSZ:      DW      0               ;;FILE SIZE (SECTORS)
642 FTY:      DB      0               ;;FILE TYPE
643 TDI:      DW      0               ;;TYPE DEPENDENT INFO
644 LADDR:    DW      0               ;;LOAD ADDR

645 ;
646 ;----- procedure FDE & buffer fields
647 ;
648 MDRV:     DB      0               ;;DEN (BIT 7), DRIVE
649 MDA:      DW      0               ;;NEXT DISK ADDR
650 MFSZ:     DW      0               ;;SECTORS REMAINING
651 MBUF:     DS      BLKSZ*DDEN      ;;PROC BUFFER (ONE SECTOR)
652 MCNT:     DW      0               ;;BYTES LEFT IN BUFFER
653 MADDR:    DW      0               ;;CURRENT BUFFER BYTE ADDR

654 ;
655 ;----- messages
656 ;
657 CRMSG:    DB      ODH
658 QUESTMSG:DB      '?',ODH
659 NESTMSG:  DB      'program name expected',ODH
660 SHELLMSG:DB      'shell error',ODH
661 RUBMSG:   DB      08H,' ',08H,0
662 PARMSG:   DB      'too many parameters',ODH
663 LINEMSG:  DB      'line too long',ODH
664 HARDMSG:  DB      'hard disk error',ODH
665 WAITMSG:  DB      NOTE,NOTE,'waiting...',0
666 ;
667 ;----- shell stack
668 ;
669             DS      48
670 BOSTACK:                          ;;BOTTOM OF STACK
671 ;
672 ;----- parameter pointers, buffer, etc.
673 ;
674 PPTRS:    DS      22              ;;11 WORDS
675 PBUF:     DS      CLSZ            ;;PARM BUFFER
676 PPADDR:   DW      0               ;;ADDR OF CUR PARM PTR
677             END
```

End Listing

68000/AIM 65

(Listing, text begins on page 12)

(See Figure 6, page 17 for meaning of fields in Listing)

```
M)=801 00 00 00 00                                  CONTROL MODE TO AUTO AND
(/) 0801 04                                         STEP
*)=200                                              RUN
(G)/
SP RW 000000 0000                   SSP FETCH
SP RW 000002 00FE
SP RW 000004 0000                   PC FETCH
SP RW 000006 0008
SP RW 000008 46FC       MOVE.W #0,SR    SWITCH TO USER PROGRAM
SP RW 00000A 0000
SP RW 00000C 4E71       NOP   FETCHED BUT NOT EXECUTED (FBNE)
UP RW 00000C 33FC
UP RW 00000E 0100       MOVE.W #$100,$FC00.L   LINK AIM SUBROUTINE READBYTE
UP RW 000010 0000
UP RW 000012 FC00
UP RW 000014 13F9
UD WW 00FC00 05         EXEC READBYTE   THE BYTE SHOWN IN THE DATA FIELD WAS
UP RW 000016 0000                       ENTERED WITH THE AIM KEYBOARD)
UP RW 000018 0000       MOVE.B 0000.L 0001.L
UP RW 00001A 0000
UD RH 000000 0504                       READ LOC. 0000
UP RW 00001C 0001
UD WL 000000 05                         WRITE LOC. 0001 (MODE CONTROL LOC.)
UP RW 00001E 7E4E71     NOP   NOW THE MODE IS CHANGED: IT IS REQUESTED A BYTE TO BE
UP RW 000020 7E4E71     NOP   STORED ON THE 68000 CONTROL INPUT LINES (LEV. 4 INT.)
SD WW 0000FC FF001E                 STACK PC LOW (CNTRL BYTE=$FF, INT. CLEARED)
1A RW FBFFFC FFM 0064                 IA            AS FUNCTION CODE=IA, WE ARE
SD WW 0000F8 FF0000                 STACK STATUS    IN MANUAL MODE; VECT.#=64
SD WW 0000FA                                        ESCAPE FROM THE PROGRAM TO
(M)=801 05 00 00 00                                 CHANGE MODE: NOW WITHOUT THE
(/) 0801 04                                         CONTROL BYTE
*)=200
(G)/
SD WW 0000FA 0000                   STACK PC HIGH
SD RW 000190 M 0000                 FETCH INT. VECT. (WE ARE IN MAN. MODE AS WE
SD RW 000192 M 4000                                   ARE OUT OF SUP. DATA SEGMENT
SP RW 004000 M 4E73     RTE         MANUAL MODE AS WE ARE OUT THE SUP. PROG. SEG
SP RW 004002 M 4E71     NOP         FBNE, MANUAL MODE
SD RW 0000F8 0000                   UNSTACK PARAMETERS
SD RW 0000FA 0000                   "
SD RW 0000FC 001E                   "
UP RW 00001E 4E71       NOP         BACK TO USER PROGRAM
UP RW 000020 4E71       NOP
UP RW 000022 4E71       NOP
UP RW 000024 4E71       NOP
UP RW 000026 4E71       NOP


M)=801 04 00 00 00                                  CONTROL MODE TO MANUAL AND
(/) 0801 06                                         STEP
*)=200                                              RUN
(G)/
SP RW 000000 M 0000                 SSP FETCH
SP RW 000002 M 1000
SP RW 000004 M 0000                 PC FETCH
SP RW 000006 M 4000
SP RW 004000 M 46FC     MOVE.W #0,SR    SWITCH TO USER PROGRAM
SP RW 004002 M 0000
SP RW 004004 M 4E71     NOP             FETCHED BUT NOT EXECUTED (FBNE)
UP RW 004004 M 42B8     CLR.L $7000.W   CLEAR LOC. $7000-$7003
UP RW 004006 M 7000
UP RW 004008 M 4E71     NOP
UD RW 007000 M FFFF                     READS THE LONG WORD (UNUSEFUL)
UD RW 007002 M FFFF
UP RW 00400A M 4E71     NOP
UD WW 007002 M 0000                     CLEAR LONG WORD
UD WW 007000 M 0000
UP RW 00400C M 4E71     NOP
UP RW 00400E M 303C     MOVE.W #$2000,D0                SET DATA REG. 0
UP RW 004010 M 2000
UP RW 004012 M C0FC     MULU.W #4,D0                    16-BIT MULTIPLICATION
UP RW 004014 M 0004
UP RW 004016 M 33C0     MOVE.W D0,$8000.L               SHOW D0 CONTENTS
UP RW 004018 M 0000
UP RW 00401A M 8000
UP RW 00401C M 4E71     NOP
UD WW 008000 M 8000                     THE RESULT OF MULU.W IS $8000
UP RW 00401E M 4E71     NOP
UP RW 004020 M 80FC     DIVU.W D0,#0    ZERO DIVIDE: WHAT HAPPENS NOW?
UP RW 004022 M 0000
UP RW 004024 M 4E71     NOP             FBNE
SD WW 000FFE M 4024                     STACK PC: ZERO DIVIDE EXCEPTION
SD WW 000FFA M 0004
SD WW 000FFC M 0000                     STACK STATUS
SD RW 000014 M 0000                     READS ZERO DIVIDE VECTOR
SD RW 000016 M 9001
SP RW 009000 M 4E73     RTE             RETURN FROM EXCEPTION
SP RW 009002 M 4E71     NOP             FBNE
SD RW 000FFA M 0004                     UNSTACK PARAMETERS
SD RW 000FFC M 0000
SD RW 000FFE M 4024
UP RW 004024 M 4E71     NOP             BACK TO USER PROGRAM
UP RW 004026 M 33C0     MOVE.W D0,$6000.L               SHOW D0 CONTENTS AFTER THE
UP RW 004028 M 0000                                     ZERO DIVIDE
UP RW 00402A M 6000
UP RW 00402C M 4E71     NOP
UD WW 006000 M 8000                                     IT DID NOT CHANGE
UP RW 00402E M 4E71     NOP
```

```
PAGE 01
PASS 2
  0000  0000            ;**********************************************
  0001  0000            ;*  MC68000 INTERFACING PROGRAM                *
  0002  0000            ;*                                             *
  0003  0000            ;*                                             *
  0004  0000            ;*  WRITTEN BY LUCA FUSINA AND CLAUDIO GRANUZZO *
  0005  0000            ;**********************************************




  0012  0000            ;IF MORE 6502 USER SUBROUTINES ARE ADDED, THEIR EXACT
  0013  0000            ;NUMBER MUST BE SPECIFIED
  0014  0000            MAXROU=2

  0016  0000            ;BOARD BASE ADDRESS
  0017  0000            BRDADD=$8000

  0019  0000            ;AIM 65 MONITOR SUBROUTINES EQUATES
  0020  0000            CRLOW=$EA13             ;OUTPUT CR,LF
  0021  0000            BLANK=$E83E             ;OUTPUT A BLANK
  0022  0000            NUMA=$EA46              ;OUTPUT A BYTE AS TWO HEX CHAR.
  0023  0000            OUTPUT=$E97A            ;OUTPUT A CHAR.
  0024  0000            RBYTE=$E3FD             ;READ TWO HEX CHAR. FROM KEYBOARD
  0025  0000            ;                        AND PACK THEM IN ONE BYTE
  0026  0000            READ=$E93C              ;READ ONE CHAR.
```

```
  0029  0000            ;USER MAY SPECIFY THESE VAR. ADDR.; IF SEGMENT ADDR.
  0030  0000            ;ARE CHANGED, THESE ADDR. MUST BE ACCORDINGLY DEFINED
  0031  0000                    *=$800
  0032  0800            SAVEAU                  ;REFER TO THE CALSUB ROUTINE
  0033  0800                    *=$500
  0034  0500            SAVEAS                  ;REFER TO THE CALSUB ROUTINE

  0037  0500            ;POINTER TO START ADDR. OF 6502 SUBR. ADDR. TABLE
  0038  0500                    *=$FA
  0039  00FA            LOCJMP

  0041  00FA            ;THE FOLLOWINGS ARE USED AS POINTER FOR
  0042  00FA            ;INDIRECT 6502 ADDRESSING
  0043  00FA                    *=*+1
  0044  00FB            F0                      ;HIGH BYTE OF A POINTER  USED
  0045  00FB            ;                        FOR 68000 TO MEMORY OPERATIONS
  0046  00FB                    *=*+1
  0047  00FC            F1                      ;LOW BYTE OF POINTER

  0049  00FC            ;THE FOLL. ARE USED TO SAVE THE 68000 ADDR.
  0050  00FC                    *=*+1
  0051  00FD            SAVEL
  0052  00FD                    *=*+1
  0053  00FE            SAVEM
  0054  00FE                    *=*+1
  0055  00FF            SAVEH
Page 02

  0058  00FF            ;MODE CONTROL VARIABLE. THE ADDR. OF THIS VAR.
  0059  00FF            ;MUST BE INSIDE USER DATA SEGMENT, SO A 68000
  0060  00FF            ;PROGRAM CAN CHANGE IT
  0061  00FF                    *=$801
  0062  0801            CONTRL

  0065  0801            ;THE FOLL. ARE THE  BOARD ADDR.; REFER TO THE TEXT
  0066  0801            ;HARDWARE DESCRIPTION FOR THE THEIR MEANING
  0067  0801                    *=BRDADD
  0068  8000            DTI
  0069  8000                    *=*+1
  0070  8001            DTH
  0071  8001                    *=*+1
  0072  8002            CIN
  0073  8002                    *=*+1
  0074  8003            COUT
  0075  8003                    *=*+1
  0076  8004            ADDL
  0077  8004                    *=*+1
  0078  8005            ADDM
  0079  8005                    *=*+1
  0080  8006            ADDH
  0081  8006                    *=*+1
  0082  8007            DOL
  0083  8007                    *=*+1
  0084  8008            DOH
  0085  8008                    *=*+1
  0086  8009            HALT
  0087  8009                    *=*+1
  0088  800A            RES
  0089  800A                    *=*+1
  0090  8008            DTACK


  0095  8008                    *=$200
  0096  0200  78        MMM6B   SEI
  0097  0201  A9FF              LDA #$FF
  0098  0203  8D0380            STA COUT        ;CLEAR 68000 INPUT SIGN.
  0099  0206  2013EA    LOOP    JSR CRLOW
  0100  0209  AD0280    WAITAS  LDA CIN
  0101  020C  2908              AND #8
  0102  020E  D0F9              BNE WAITAS      ;WAIT FOR ADDR. STROBE
  0103  0210  AD0108            LDA CONTRL
  0104  0213  2908              AND #8          ;SPEEDY MODE CONTROL BIT
  0105  0215  F003              BEQ OUTFC
  0106  0217  4C2E03            JMP FAST

  0109  021A            ;OUTPUT 68000 FUNCTION CODE
  0110  021A            ;U1(UNDEFINED),U2(UNDEFINED),UP(USER
  0111  021A            ;PROGRAM),SP(SUPERVISOR PROGRAM),
  0112  021A            ;UD(USER DATA),SD(SUPERVISOR DATA),
  0113  021A            ;U3(UNDEFINED),IA(INTERRUPT ACK.)
  0114  021A  AD0280    OUTFC   LDA CIN
Page 03
  0115  021D  29E0              AND #%11100000
  0116  021F  4A                LSR A
  0117  0220  4A                LSR A
  0118  0221  4A                LSR A
  0119  0222  4A                LSR A
  0120  0223  AA                TAX
  0121  0224  BDE803            LDA TABFC,X
  0122  0227  207AE9            JSR OUTPUT
  0123  022A  BDE903            LDA TABFC+1,X
  0124  022D  207AE9            JSR OUTPUT
  0125  0230  203EE8            JSR BLANK

  0128  0233            ;OUTPUT 68000 R/W CODE
  0129  0233            ;WW(WRITE WORD),RW(READ WORD),WH(WRITE HIGH
  0130  0233            ;BYTE),WL(WRITE LOW BYTE),RL(READ LOW BYTE),
  0131  0233            ;RH(READ HIGH BYTE),UN(UNDEFINED)
  0132  0233  AD0280    OUTRWL  LDA CIN
  0133  0236  2907              AND #7
  0134  0238  0A                ASL A
  0135  0239  AA                TAX
  0136  023A  BDD803            LDA TABRWL,X
  0137  023D  207AE9            JSR OUTPUT
  0138  0240  BDD903            LDA TABRWL+1,X
  0139  0243  207AE9            JSR OUTPUT
  0140  0246  203EE8            JSR BLANK

  0143  0249            ;OUTPUT 68000 ADDRESSES
  0144  0249            ;WE MULTIPLY BY TWO TO CONVERT FROM
  0145  0249            ;68000 WORD ADDR. TO BYTE ADDR.
  0146  0249  AD0480    OUTADD  LDA ADDL
  0147  024C  0A                ASL A
  0148  024D  85FD              STA SAVEL
  0149  024F  AD0580            LDA ADDM
  0150  0252  2A                ROL A
  0151  0253  85FE              STA SAVEM
  0152  0255  AD0680            LDA ADDH
  0153  0258  2A                ROL A
```

```
0154  0259  85FF            STA SAVEH
0155  025B  2046EA          JSR NUMA
0156  025E  A5FE            LDA SAVEM
0157  0260  2046EA          JSR NUMA
0158  0263  A5FD            LDA SAVEL
0159  0265  2046EA          JSR NUMA
0160  0268  203EE8          JSR BLANK

0163  026B  AD0108  INCMD   LDA CONTRL
0164  026E  2901            AND #1          ;WITH THIS BIT SET A BYTE IS
0165  0270        ;                         ENTERED FROM KEYBOARD
0166  0270        ;                         TO CONTROL THE 68000 INPUT SGN.
0167  0270  F006            BEQ NOCMD
0168  0272  20FDE3  CMD     JSR RBYTE
0169  0275  8D0380          STA COUT
0170  0278  AD0108  NOCMD   LDA CONTRL
0171  027B  2902            AND #2          ;MANUAL OR AUTO MODE CNTR. BIT
0172  027D  F02D            BEQ AUTO
```

Page 04

```
0175  027F        ;MANUAL MODE: READ FROM KEYBOARD,
0176  027F        ;MEMORY WRITE TO DISPLAY/PRINTER ONLY
0177  027F  A94D  MANUAL  LDA #$4D
0178  0281  207AE9          JSR OUTPUT      ;OUTPUT 'M'
0179  0284  203EE8          JSR BLANK
0180  0287  AD0280          LDA CIN
0181  028A  2901            AND #1          ;TEST 68000 R/W
0182  028C  D00F            BNE READM

0184  028E        ;IF WRITE OUTPUT 68000 DATA BUS
0185  028E  AD0180  WRITEN  LDA DIH
0186  0291  2046EA          JSR NUMA
0187  0294  AD0080          LDA DIL
0188  0297  2046EA          JSR NUMA
0189  029A  4C1703          JMP TSER

0191  029D        ;IF READ, GET TWO BYTES FROM KEYBOARD AND PUT THEM
0192  029D        ;ON THE 68000 INPUT DATA LATCHES
0193  029D  20FDE3  READM   JSR RBYTE
0194  02A0  8D0880          STA DOH
0195  02A3  20FDE3          JSR RBYTE
0196  02A6  8D0780          STA DOL
0197  02A9  4C1703          JMP TSER

0200  02AC        ;AUTO MODE: R/W FROM MEMORY
0201  02AC  AD0280  AUTO    LDA CIN

0203  02AF        ;MEMORY SEGMENTATION SECTION: IF OVERFLOW
0204  02AF        ;ERROR OR INTA STATE SWITCH TO MANUAL MODE
0205  02AF  29E0  FCADD   AND #%11100000
0206  02B1  4A            LSR A
0207  02B2  4A            LSR A
0208  02B3  4A            LSR A
0209  02B4  4A            LSR A
0210  02B5  4A            LSR A
0211  02B6  AA            TAX
0212  02B7  E007          CPX #7
0213  02B9  F0C4          BEQ MANUAL      ;68000 HAS ACK. AN INTERRUPT
0214  02BB  A5FF          LDA SAVEH
0215  02BD  D0C0          BNE MANUAL      ;SEGMENT GREATER THAN 64K.
0216  02BF  A5FE          LDA SAVEM
0217  02C1  C9FC          CMP #$FC        ;6502 SUBROUTINE CALL IF 68000
0218  02C3        ;                       ADDR. IS EQUAL TO $00FC00
0219  02C3  D003            BNE LABO
0220  02C5  4CA203          JMP CALSUB
0221  02C8  DD0004  LABO    CMP MAXADD,X
0222  02CB  B0B2            BCS MANUAL      ;ERR. IF SEGMENT IS
0223  02CD        ;                         OUT OF ITS RANGE
0224  02CD  18              CLC
0225  02CE  7DFB03          ADC FCTAB,X     ;ADD SEG. OFFSET TO FFFECTIVE
0226  02D1        ;                       68000 ADDR., SO TO SELECT THE
0227  02D1        ;                       RIGHT MEMORY LOCATIONS IN THE
0228  02D1        ;                       AIM MEMORY SPACE
0229  02D1  85FC            STA F1
0230  02D3  A5FD            LDA SAVEL
```

Page 05

```
0231  02D5  85F8            STA F0
0232  02D7  AD0280          LDA CIN
0233  02DA  2901            AND #1          ;TEST 68000 R/W
0234  02DC  D025            BNE READA

0236  02DE        ;AUTO WRITE IN MEMORY
0237  02DE  AD0280  WRITA   LDA CIN
0238  02E1  2902            AND #2          ;TEST 68000 UPPER DATA STROBE
0239  02E3  D00A            BNE NOUDS

0241  02E5        ;WRITE UPPER DATA BYTE
0242  02E5  A000    WRIUDS  LDY #00
0243  02E7  AD0180          LDA DIH
0244  02EA  91F8            STA (F0),Y
0245  02EC  2046EA          JSR NUMA
0246  02EF  AD0280  NOUDS   LDA CIN
0247  02F2  2904            AND #4          ;TEST 68000 LOWER DATA STROBE
0248  02F4  D00A            BNE NOLDS

0250  02F6        ;WRITE 68000 LOWER DATA BYTE
0251  02F6  A001    WRILDS  LDY #1
0252  02F8  AD0080          LDA DIL
0253  02FB  91F8            STA (F0),Y
0254  02FD  2046EA          JSR NUMA
0255  0300  4C1703  NOLDS   JMP TSER

0257  0303        ;READ A WORD FROM MEMORY
0258  0303  A000    READA   LDY #0
0259  0305  B1F8            LDA (F0),Y
0260  0307  8D0880          STA DOH
0261  030A  2046EA          JSR NUMA
0262  030D  A001            LDY #1
0263  030F  B1F8            LDA (F0),Y
0264  0311  8D0780          STA DOL
0265  0314  2046EA          JSR NUMA

0268  0317        ;IF STEP MODE IS SELECTED, WAIT FOR A CHAR. FROM
0269  0317        ;KEYBOARD: IF IT IS NOT A CR DO NOT ISSUE
0270  0317        ;DTACK, AND REPEAT THE CICLE
0271  0317  AD0108  TSER    LDA CONTRL
0272  031A  2904            AND #4          ;STEP MODE CONTROL BIT
0273  031C  F00A            BEQ DTA
0274  031E  203CE9          JSR READ
0275  0321  C90D            CMP #$0D
0276  0323  F003            BEQ DTA
0277  0325  4C0602          JMP LOOP
0279  0328        ;ISSUE DTACK
0280  0328  AD0B80  DTA     LDA DTACK
0281  032B  4C0602          JMP LOOP

0284  032E        ;FAST AUTO MODE: NO OUTPUT IS PERFORMED

0286  032E        ;SAVE 68000 ADDR.
0287  032E  AD0480  FAST    LDA ADDL        ;MULTIPLY BY TWO
0288  0331  0A              ASL A
```

Page 06

```
0289  0332  85F8            STA F0
0290  0334  AD0580          LDA ADDM
0291  0337  2A              ROL A
0292  0338  85FE            STA SAVEM
0293  033A  AD0680          LDA ADDH
0294  033D  2A              ROL A
0295  033E  F003            BEQ FAST1       ;SKIP IF SEG. IS LESS THAN 64K
0296  0340  4C1A02  OUTF1   JMP OUTFC        ;ERR.: OUTPUT CYCLE STAT.

0298  0343        ;MEMORY SEGMENTATION SECTION: ON ERROR
0299  0343        ;STANDARD CYCLE HANDLING IS ENTERED, AND
0300  0343        ;FAST MODE IS RE-ENTERED NEXT CYCLE
0301  0343  AD0280  FAST1   LDA CIN
0302  0346  29E0            AND #$E0        ;68000 FUNCTION CODE
0303  0348  4A              LSR A
0304  0349  4A              LSR A
0305  034A  4A              LSR A
0306  034B  4A              LSR A
0307  034C  4A              LSR A
0308  034D  AA              TAX
0309  034E  E007            CPX #7
0310  0350  F0EE            BEQ OUTF1       ;SKIP IF INTA
0311  0352  A5FE            LDA SAVEM
0312  0354  C9FC            CMP #$FC        ;6502 SUBROUTINES CALL
0313  0356  F04A            BEQ CALSUB
0314  0358  A5FE            LDA SAVEM
0315  035A  DD0004          CMP MAXADD,X
0316  035D  B0E1            BCS OUTF1       ;OUT OF SEG. RANGE ERR.
0317  035F  18              CLC
0318  0360  7DFB03          ADC FCTAB,X     ;ADD SEG. OFFSET
0319  0363  85FC            STA F1
0320  0365  AD0280          LDA CIN
0321  0368  2901            AND #1          ;TEST 68000 R/W
0322  036A  D022            BNE READF

0324  036C        ;FAST MEMORY WRITE
0325  036C  AD0280  WRITF   LDA CIN
0326  036F  2902            AND #2          ;68000 UDS
0327  0371  D007            BNE NOUDSF
0328  0373  A000    WRFUDS  LDY #0
0329  0375  AD0180          LDA DIH
0330  0378  91F8            STA (F0),Y
0331  037A  AD0280  NOUDSF  LDA CIN
0332  037D  2904            AND #4          ;68000 LDS
0333  037F  D007            BNE NOLDSF
0334  0381  A001    WRFLDS  LDY #1
0335  0383  AD0080          LDA DIL
0336  0386  91F8            STA (F0),Y
0337  0388  AD0B80  NOLDSF  LDA DTACK
0338  038B  4C0902          JMP WAITAS

0340  038E        ;FAST MEMORY READ
0341  038E  A000    RFADF   LDY #0
0342  0390  B1F8            LDA (F0),Y
0343  0392  8D0880          STA DOH
0344  0395  A001            LDY #1
0345  0397  B1F8            LDA (F0),Y
0346  0399  8D0780          STA DOL
```

Page 07

```
0347  039C  AD0B80          LDA DTACK
0348  039F  4C0902          JMP WAITAS

0351  03A2        ;6502 SUBROUTINES HANDLER.  68000 UPPER
0352  03A2        ;DATA BYTE IS THE INDEX TO THE SUBROUTINES
0353  03A2        ;ADDRESS TABLE: 68000 LOWER
0354  03A2        ;DATA BYTE IS LOADED IN THE 6502 ACC.
0355  03A2        ;ON RETURN THE 6502 ACC. IS COPIED IN
0356  03A2        ;TWO LOCATIONS SPECIFIED BY THE USER:
0357  03A2        ;SAVEAU IS LOCATED IN THE USER DATA SEGMENT;
0358  03A2        ;SAVEAS IS LOCATED IN THE SUPERVISOR
0359  03A2        ;DATA SEGMENT
0360  03A2  AD0280  CALSUB  LDA CIN
0361  03A5  2901            AND #1          ;TEST 68000 R/W BECAUSE
0362  03A7  F003            BEQ CALSU1      ;ONLY WRITE IS ALLOWED
0363  03A9  4C7F02  ERR     JMP MANUAL
0364  03AC  AD0180  CALSU1  LDA DIH
0365  03AF  C902            CMP #MAXROU     ;SUBR. ADDR. TABLE OVERFLOW
0366  03B1  B0F6            BCS ERR
0367  03B3  0A              ASL A
0368  03B4  AA              TAX
0369  03B5  BDD403          LDA TABSUB,X
0370  03B8  85FA            STA LOCJMP
0371  03BA  BDD503          LDA TABSUB+1,X
0372  03BD  85F8            STA LOCJMP+1
0373  03BF  AD0080          LDA DIL
0374  03C2  20D103          JSR JMPSUB
0375  03C5  8D0008          STA SAVEAU
0376  03C8  8D0005          STA SAVEAS
0377  03CB  AD0B80          LDA DTACK
0378  03CE  4C0602          JMP LOOP

0380  03D1  6CFA00  JMPSUB  JMP (LOCJMP)


0384  03D4        ;6502 SUBR. TABLE
0385  03D4  46EA    TABSUB  .WORD NUMA
0386  03D6  FDE3            .WORD RBYTE

0388  03D8        ;68000 R/W CODES
0389  03D8  5757    TABRWL  .BYTE 'WWRWWLRLWHRHUNUN'

0391  03E8        ;68000 FUNCTION CODES
0392  03E8  5531    TABFC   .BYTE 'U1U2UPSPUDSDU3IA'

0394  03F8        ;SEGMENT START ADDR., EXPRESSED IN PAGES OF 256
0395  03F8        ;BYTES EACH. USER MAY CHANGE THESE VALUES; IF
0396  03F8        ;HE DOES SO HE HAS TO CHANGE ALL THE RELATED
0397  03F8        ;VARIABLE ADDRESSES (SEE TOP OF PROGRAM).
0398  03F8        ;HE MAY HAVE TO CHANGE VALUES IN THE MAXADD
```

(Continued on next page)

68000/AIM 65

(Listing continued, text begins on page 12)

```
0399  03F8            ;TABLE,TOO
0400  03F8  00        FCTAB   .BYTE 0         ;UNDEFINED
0401  03F9  00                .BYTE 0         ;UNDEFINED
0402  03FA  0C                .BYTE $0C       ;USER PROGRAM
0403  03FB  06                .BYTE 6         ;SUPERVISOR PROGRAM
0404  03FC  08                .BYTE 8         ;USER DATA

Page 08

0405  03FD  05                .BYTE 5         ;SUPERVISOR DATA
0406  03FE  00                .BYTE 0         ;UNDEFINED
0407  03FF  00                .BYTE 0         ;(INTA)


0410  0400            ;NUMBER OF PAGES AVAILABLE TO EACH SEGMENT.
0411  0400            ;USER MAY CHANGE THESE VALUES; HOWEVER THEY
0412  0400            ;MUST BE CONSISTENT WITH THE SEGMENTS START
0413  0400            ;ADDRESSES JUST DEFINED
0414  0400  00        MAXADD  .BYTE 0         ;UNDEFINED
0415  0401  00                .BYTE 0         ;UNDEFINED
0416  0402  04                .BYTE 4         ;USER PROGRAM (1K)
0417  0403  02                .BYTE 2         ;SUP. PROGRAM (512 BYTES)
0418  0404  04                .BYTE 4         ;USER DATA (1K)
0419  0405  01                .BYTE 1         ;SUP. DATA (256 BYTES)
0420  0406  00                .BYTE 0         ;UNDEFINED
0421  0407  00                .BYTE 0         ;(INTA)


0424  0408            ;END OF PROGRAM

0426  0408                    .END

ERRORS- 0000
```

```
MAXROU  A: 0002   BRDADD  A: 8000   CRLOW   A: EA13   BLANK   A: E83E
NUMA    A: EA44   OUTPUT  A: E97A   RBYTE   A: E3ED   READ    A: E93C
SAVEAU  A: 0800   SAVEAS  A: 0500   LOCJMP  A: 00FA   F0      A: 00FB
F1      A: 00FC   SAVEL   A: 00FD   SAVEH   A: 00FE   SAVEH   A: 00FF
CONTRL  A: 8001   DIL     A: 8000   DIH     A: 8001   CIN     A: 8002
COUT    A: 8003   ADDL    A: 8004   ADDH    A: 8005   ADDH    A: 8006
DCL     A: 8007   DOH     A: 8008   HALT    A: 8009   RES     A: 800A
DTACK   A: 800B   MMM68   A: 0200   LOOP    A: 0206   WAITAS  A: 0209
OUTEC   A: 021A   OUTRWL  A: 0233   OUTADD  A: 0249   INCMD   A: 026B
CMD     A: 0272   NOCMD   A: 0278   MANUAL  A: 027F   WRITEM  A: 028E
READM   A: 029D   AUTO    A: 02AC   FCADD   A: 02AF   LABO    A: 02C8
WRITA   A: 02DE   WRTUDS  A: 02E5   NOUDS   A: 02EF   WRILDS  A: 02F6
NOLDS   A: 0300   READA   A: 0303   TSER    A: 0317   DTA     A: 0328
FAST    A: 032E   OUTF1   A: 0340   FAST1   A: 0343   WRITF   A: 036C
WRFUDS  A: 0373   NOUDSF  A: 037A   WRFLDS  A: 0381   NOLDSF  A: 0388
READI   A: 038E   CALSUB  A: 03A2   ERR     A: 03A9   CALSU1  A: 03AC
JMPSU   A: 03D1   TABSUB  A: 03D4   TABRWL  A: 03DB   TABFC   A: 03E8
FCTAB   A: 03F8   MAXADD  A: 0400
```

End Listing

Screen Oriented Editor

See bug fixes, DDJ Bound Volume page 193.

```
b:ed.c                  May 11, 1981                    page 1

/* Include file for screen editor
 *
 * Source: ed.c
 * Version: Feb. 10, 1980.
 */

#include ed0.c          /* constant (system) globals */
#include ed1.ccc        /* user defined globals */
#include ed2.c          /* main program */
#include ed3.c          /* command mode commands */
#include ed4.c          /* edit mode module */
#include ed5.c          /* output format module */
#include ed6.ccc        /* display screen module */
#include ed7.c          /* prompt line module */
#include ed8.c          /* operating system module */
#include ed9.c          /* utility routines */
#include ed10.c         /* memory buffer module */
#include c80lib         /* machine language library */
```

```
   ed0.c                September 8, 1981              page 1

/* Screen editor:  non-user defined globals
 *
 * Source: ed0.c
 * Version: June 19, 1980.
 */


/* Define global constants */

/* Define constants describing a text line */

#define MAXLEN  133     /* max chars per line */
#define MAXLEN1 134     /* MAXLEN + 1 */

/* Define operating system constants */

#define SYSFNMAX 15     /* CP/M file name length + 1 */

/* Define misc. constants */

#define EOS     0       /* code sometimes assumes \0 */
#define OK      1
#define ERR     -1      /* error.  must be <0 */
#define EOF     -2      /* end of file.  must be <0 */
#define YES     1       /* must be nonzero */
#define NO      0
#define CR      13      /* carriage return */
#define LF      10      /* line feed */
#define TAB     9       /* tab character */
#define HUGE    32000   /* practical infinity */
```

```
ed1.ccc                 September 8, 1981               page 1


/*
Screen editor:  special key definitions
Source:  ed1.ccc
This file was created by the configuration program:
Version 2:  September 6, 1981.
*/

/*
Define which keys are used for special edit functions.
*/

#define UP1 10
#define DOWN1 13
#define UP2 21
#define DOWN2 4
#define LEFT1 8
#define RIGHT1 18
#define INS1 14
#define EDIT1 5
#define ESC1 27
#define DEL1 127
#define ZAP1 26
#define ABT1 24
#define SPLT1 19
#define JOIN1 16

/*
Define length and width of screen and printer.
*/
#define SCRNW 64
#define SCRNW1 63
#define SCRNL 16
#define SCRNL1 15
#define SCRNL2 14
#define LISTW 132
```

```
ed2.c                   September 8, 1981               page 1

/* Screen editor:  main program
 *
 * Source:  ed2.c
 * Version: September 5, 1981.
 */

/* define signon message */

#define SIGNON "small-c editor, version 2:  September 5, 1981."

/* the main program dispatches the routines that
 * handle the various modes.
 */

#define CMNDMODE 1      /* enter command mode flag */
#define INSMODE  2      /* enter insert modes flag */
#define EDITMODE 3      /* enter edit mode flag */
#define EXITMODE 4      /* exit editor flag */

main()
{
int mode;
        /* fmt output by default goes to screen */
        fmtassn(NO);
        /* set tabs, clear the screen and sign on */
        fmtset(8);
        outclr();
        outxy(0,SCRNL1);
        message(SIGNON);
        outxy(0,1);
        /* clear filename[] for save(), resave() */
        name("");
        /* clear the main buffer */
        bufnew();
        /* start off in command mode */
        mode=CMNDMODE;
        /* get null line 1 for edit() */
        edgetln();
        while(1){
                if (mode==EXITMODE) {
                        break;
                }
                else if (mode==CMNDMODE) {
                        mode=command();
                }
                else if (mode==EDITMODE) {
                        mode=edit();
                }
                else if (mode==INSMODE) {
                        mode=insert();
                }
                else {
                        syserr("main: no mode");
                        mode=EDITMODE;
                }
        }
}

/*
 * handle edit mode.
 * dispatch the proper routine based on one-character commands.
 */
```

```
ed2.c                   September 8, 1981               page 2


edit()
{
char buffer[SCRNW1];
int v;
int x,y;
char c;
        /* we can't do edgetln() or edgo() here because
         * those calls reset the cursor.
         */
        pmtedit();
        while(1){
                /* get command */
                c=tolower(syscin());
                if ((c==ESC1)|(c=='c')) {
                        /* enter command mode. */
                        return(CMNDMODE);
                }
```

```
        else if ((c==INS1)|(c=='i')) {
                /* enter insert mode */
                return(INSMODE);
        }
        else if (special(c) == YES) {
                if ((c == UP1)|(c == DOWN1)) {
                        return(INSMODE);
                }
                else {
                        continue;
                }
        }
        else if (control(c)==YES) {
                continue;
        }
        else if (c==' ') {
                edright();
                pmtcol();
        }
        else if (c=='b') {
                edbegin();
                pmtcol();
        }
        else if (c=='d') {
                /* scroll down */
                pmtmode("edit: scroll");
                while (bufnrbot()==NO) {
                        if (chkkey()==YES) {
                                break;
                        }
                        if (eddn()==ERR) {
                                break;
                        }
                }
                pmtedit();
        }
        else if (c=='e') {
                edend();
                pmtcol();
        }
        else if (c=='g') {
                /* save x,y in case don't get number */
                x=outgetx();
                y=outgety();

ed2.c                   September 8, 1981               page 3

                pmtcmnd("edit: goto: ",buffer);
                if(number(buffer,&v)) {
                        edgo(v,0);
                }
                else {
                        outxy(x,y);
                }
                pmtedit();
        }
        else if (c=='k') {
                pmtmode("edit: kill");
                c=syscin();
                if ((special(c)==NO) &
                    (control(c)==NO)) {
                        edkill(c);
                }
                pmtedit();
        }
        else if (c=='s') {
                pmtmode("edit: search");
                c=syscin();
                if ((special(c)==NO) &
                    (control(c)==NO)) {
                        edsrch(c);
                }
                pmtedit();
        }
        else if (c=='u') {
                /* scroll up */
                pmtmode("edit: scroll");
                while (bufattop()==NO) {
                        if (chkkey()==YES) {
                                break;
                        }
                        if (edup()==ERR) {
                                break;
                        }
                }
                pmtedit();
        }
        else if (c=='x') {
                pmtmode("edit: eXchange");
                c=syscin();
                if ((special(c)==NO) &
                    (control(c)==NO)) {
                        edchng(c);
                }
                pmtedit();
        }
        /* do nothing if command not found */
    }
}

/* insert mode.
 * in this mode the UP1, UP2 keys reverse their roles,
 * as do the DOWN1, and DOWN2 keys.
 */

insert()
{
char c;
        pmtmode("insert");

ed2.c                   September 8, 1981               page 4

        while (1) {
                /* get command */
                c=syscin();
                if (c==ESC1) {
                        /* enter command mode */
                        return(CMNDMODE);
                }
                else if (c==EDIT1) {
                        /* enter edit mode */
                        return(EDITMODE);
                }
                else if (c==INS1) {
                        /* do nothing */
                        ;
                }
                else if (special(c)==YES) {
                        if ((c == UP2)|(c == DOWN2)) {
                                return(EDITMODE);
                        }
                        else {
                                continue;
                        }
                }
                else if (control(c)==YES) {
                        /* ignore non-special control chars */
                        continue;
                }
                else {
                        /* insert one char in line */
                        edins(c);
                        pmtcol();
                }
        }
}

/* return YES if c is a control char */

control(c) char c;
{
        if (c==TAB) {
                return(NO);     /* tab is regular */
        }
        else if (c>=127) {
                return(YES);    /* del or high bit on */
        }
        else if (c<32) {
                return(YES);    /* control char */
        }
        else {
                return(NO);     /* normal */
        }
}

/*
 * handle the default actions of all special keys.
 * return YES if c is one of the keys.
 */

special(c) char c;
{
int k;
        if (c==JOIN1) {

ed2.c                   September 8, 1981               page 5

                edjoin();
                pmtline();
                return(YES);
        }
        if (c==SPLT1) {
                edsplit();
                pmtline();
                return(YES);
```

Screen Oriented Editor

(Listing continued, text begins on page 18)

```
        }
        if (c==ABT1) {
                edabt();
                pmtcol();
                return(YES);
        }
        else if (c==DEL1) {
                eddel();
                pmtcol();
                return(YES);
        }
        else if (c==ZAP1) {
                edzap();
                pmtline();
                return(YES);
        }
        else if (c==UP2) {
                /* move up */
                edup();
                pmtline();
                return(YES);
        }
        else if (c==UP1) {
                /* insert up */
                ednewup();
                pmtline();
                return(YES);
        }
        else if (c==DOWN2) {
                /* move down */
                eddn();
                pmtline();
                return(YES);
        }
        else if (c==DOWN1) {
                /* insert down */
                ednewdn();
                pmtline();
                return(YES);
        }
        else if (c==LEFT1) {
                edleft();
                pmtcol();
                return(YES);
        }
        else if (c==RIGHT1) {
                edright();
                pmtcol();
                return(YES);
        }
        else {
                return(NO);
        }
}
```

ed2.c September 8, 1981 page 6

```
/*
 * command() dispatches command routines while
 * in command mode.
 */

command()
{
int v;
char c;
char args[SCRNW1];
char *argp;
int topline;
int ypos;
int oldline;
int k;
        /* command mode commands may move the current line.
         * command mode must save the current line on entry
         * and restore it on exit.
         */
        edrepl();
        /* remember how the screen was drawn on entry */
        oldline=bufln();
        ypos=outgety();
        topline=oldline-ypos+1;
        while(1) {
                outxy(0,SCRNL1);
                fmtcrlf();
                pmtmode("command:");
                getcmnd(args,0);
                fmtcrlf();
                pmtline();
                c=args[0];
                if ((c==EDIT1)|(c==INS1)) {
                        /* redraw screen */
                        if (oldline==bufln()) {
                                /* get current line */
                                edgetln();
                                /* redraw old screen */
                                bufout(topline,1,SCRNL1);
                                outxy(0,ypos);
                        }
                        else {
                                /* update line and screen */
                                edgo(bufln(),0);
                        }
                        if (c==EDIT1) {
                                return (EDITMODE);
                        }
                        else {
                                return (INSMODE);
                        }
                }
                else if (tolower(args[0])=='g'){
                        argp=skipbl(args+1);
                        if (argp[0]==EOS) {
                                edgo(oldline,0);
                                return(EDITMODE);
                        }
                        else if (number(argp,&v)==YES) {
                                edgo(v,0);
                                return(EDITMODE);
```

ed2.c September 8, 1981 page 7

```
                        }
                        else {
                                message("bad line number");
                        }
                }
                else if (lookup(args,"append")) {
                        append(args);
                }
                else if (lookup(args,"change")) {
                        change(args);
                }
                else if (lookup(args,"clear")) {
                        clear();
                }
                else if (lookup(args,"delete")) {
                        delete(args);
                }
                else if (lookup(args,"dos")) {
                        if (chkbuf()==YES) {
                                return (EXITMODE);
                        }
                }
                else if (lookup(args,"find")) {
                        if ((k = find()) >= 0) {
                                edgo(bufln(),k);
                                return(EDITMODE);
                        }
                        else {
                                /* get current line */
                                bufgo(oldline);
                                edgetln();
                                /* stay in command mode */
                                message("pattern not found");
                        }
                }
                else if (lookup(args,"list")) {
                        list(args);
                }
                else if (lookup(args,"load")) {
                        load(args);
                }
                else if (lookup(args,"name")) {
                        name(args);
                }
                else if (lookup(args,"resave")) {
                        resave();
                }
                else if (lookup(args,"save")) {
                        save();
                }
                else if (lookup(args,"search")) {
                        search(args);
                }
                else if (lookup(args,"tabs")) {
                        tabs(args);
                }
                else {
                        message("command not found");
                }
```

```
        }
}
```

ed2.c September 8, 1981 page 8

```
/* return YES if line starts with command */

lookup(line,command) char *line, *command;
{
        while(*command) {
                if (tolower(*line++)!=*command++) {
                        return(NO);
                }
        }
        if((*line==EOS)|(*line==' ')|(*line==TAB)) {
                return(YES);
        }
        else {
                return(NO);
        }
}

/* get next command into argument buffer */

getcmnd(args,offset) char *args; int offset;
{
int j,k;
char c;
        outxy(offset,outgety());
        outdeol();
        k=0;
        while ((c=syscin())!=CR) {
                if ((c==EDIT1)|(c==INS1)) {
                        args[0]=c;
                        return;
                }
                if ((c==DEL1)|(c==LEFT1)) {
                        if (k>0) {
                                outxy(offset,outgety());
                                outdeol();
                                k--;
                                j=0;
                                while (j<k) {
                                        outchar(args[j++]);
                                }
                        }
                }
                else if (c==ABT1) {
                        outxy(offset,outgety());
                        outdeol();
                        k=0;
                }
                else if ((c!=TAB)&((c<32)|(c==127))) {
                        /* do nothing */
                        continue;
                }
                else {
                        if ((k+offset)<SCRNW1) {
                                args[k++]=c;
                                outchar(c);
                        }
                }
        }
        args[k]=EOS;
}
```

ed3.c September 8, 1981 page 1

```
/*
 * Screen editor:  command mode commands
 *
 * Source: ed3.c
 * Version: September 5, 1981.
 *
 */

/* data global to these routines */

char filename[SYSFNMAX];


/* append command.
 * load a file into main buffer at current location.
 * this command does NOT change the current file name.
 */

append(args) char *args;
{
char buffer[MAXLEN];            /* disk line buffer */
int file;
int n;
int topline;
char locfn[SYSFNMAX];           /* local file name */
        /* get file name which follows command */
        if (name1(args,locfn)==ERR) {
                return;
        }
        if (locfn[0]==EOS) {
                message("no file argument");
                return;
        }
        /* open the new file */
        if ((file=sysopen(locfn,"r"))==ERR) {
                message("file not found");
                return;
        }
        /* read the file into the buffer */
        while ((n=readline(file,buffer,MAXLEN))>=0) {
                if (n>MAXLEN) {
                        message("line truncated");
                        n=MAXLEN;
                }
                if (bufins(buffer,n)==ERR) {
                        break;
                }
                if (bufdn()==ERR) {
                        break;
                }
        }
        /* close the file */
        sysclose(file);
        /* redraw the screen so topline will be at top
         * of the screen after command() does a CR/LF.
         */
        topline=max(1,bufln()-SCRNL2);
        bufout(topline,2,SCRNL2);
        bufgo(topline);
}

/* global change command */
```

ed3.c September 8, 1981 page 2

```
change(args) char *args;
{
char oldline[MAXLEN1];          /* reserve space for EOS */
char newline[MAXLEN1];
char oldpat[MAXLEN1];
char newpat[MAXLEN1];
int from, to, col, n, k;
        if (get2args(args,&from,&to)==ERR) {
                return;
        }
        /* get search and change masks into oldpat, newpat */
        fmtsout("search mask ?  ",0);
        getcmnd(oldpat,15);
        fmtcrlf();
        if (oldpat[0]==EOS) {
                return;
        }
        pmtline();
        fmtsout("change mask ?  ",0);
        getcmnd(newpat,15);
        fmtcrlf();
        /* make substitution for lines between from, to */
        while (from<=to) {
                if (chkkey()==YES) {
                        break;
                }
                if (bufgo(from++)==ERR) {
                        break;
                }
                if (bufatbot()==YES) {
                        break;
                }
                n=bufgetln(oldline,MAXLEN);
                n=min(n,MAXLEN);
                oldline[n]=EOS;
                /* '^' anchors search */
                if (oldpat[0]=='^') {
                        if (amatch(oldline,oldpat+1,0)==YES) {
                                k=replace(oldline,newline,
                                        oldpat+1,newpat,0);
                                if (k==ERR) {
                                        return;
                                }
                                fmtcrlf();
                                putdec(bufln(),5);
                                fmtsout(newline,5);
                                outdeol();
                                bufrepl(newline,k);
                        }
```

(Continued on next page)

Screen Oriented Editor

(Listing continued, text begins on page 18)

```
                continue;
        }
        /* search oldline for oldpat */
        col=0;
        while (col<n) {
                if (amatch(oldline,oldpat,col++)==YES){
                        k=replace(oldline,newline,
                                oldpat,newpat,col-1);
                        if (k==ERR) {
                                return;
                        }
                        fmtcrlf();
```



```
                        putdec(bufln(),5);
                        fmtsout(newline,5);
                        outdeol();
                        bufrepl(newline,k);
                        break;
                }
        }
    }
    fmtcrlf();
}

/* clear main buffer and file name */

clear()
{
    /* make sure it is ok to clear buffer */
    if (chkbuf()==YES) {
        filename[0]=0;
        pmtfile("");
        outclr();
        outxy(0,SCRNL1);
        bufnew();
        message("buffer cleared");
    }
}

/* multiple line delete command */

delete(args) char *args;
{
int from, to;
    if (get2args(args,&from,&to)==ERR) {
        return;
    }
    if (from>to) {
        return;
    }
    /* go to first line to be deleted */
    if (bufgo(from)==ERR) {
        return;
    }
    /* delete all line between from and to */
    if (bufdeln(to-from+1)==ERR) {
        return;
    }
    /* redraw the screen */
    bufout(bufln(),1,SCRNL1);
}

/* search all lines below the current line for a pattern
 * return -1 if pattern not found.
 * otherwise, return column number of start of pattern.
 */

find()
{
    return(search1(bufln()+1,HUGE,YES));
}

/* list lines to list device */

list(args) char *args;
```



```
{
char linebuf[MAXLEN1];
int n;
int from, to, line, oldline;
    /* save the buffer's current line */
    oldline=bufln();
    /* get starting, ending lines to print */
    if (get2args(args,&from,&to)==ERR) {
        return;
    }
    /* print lines one at a time to list device */
    line=from;
    while (line<=to) {
        /* make sure prompt goes to console */
        fmtassn(NO);
        /* check for interrupt */
        if (chkkey()==YES) {
            break;
        }
        /* print line to list device */
        fmtassn(YES);
        if (bufgo(line++)!=OK) {
            break;
        }
        if (bufatbot()) {
            break;
        }
        n=bufgetln(linebuf,MAXLEN1);
        n=min(n,MAXLEN);
        linebuf[n]=CR;
        fmtsout(linebuf,0);
        fmtcrlf();
    }
    /* redirect output to console */
    fmtassn(NO);
    /* restore cursor */
    bufgo(oldline);
}

/* load file into buffer */

load (args) char *args;
{
char buffer[MAXLEN];    /* disk line buffer */
char locfn [SYSFNMAX];  /* file name until we check it */
int n;
int file;
int topline;
    /* get filename following command */
    if (name1(args,locfn)==ERR) {
        return;
    }
    if (locfn[0]==EOS) {
        message("no file argument");
        return;
    }
    /* give user a chance to save the buffer */
    if (chkbuf()==NO) {
        return;
    }
    /* open the new file */
    if ((file=sysopen(locfn,"r"))==ERR) {
```



```
        message("file not found");
        return;
    }
    /* update file name */
    syscopfn(locfn, filename);
    pmtfile(filename);
    /* clear the buffer */
    bufnew();
    /* read the file into the buffer */
    while ((n=readline(file,buffer,MAXLEN))>=0) {
        if (n>MAXLEN) {
            message("line truncated");
            n=MAXLEN;
        }
        if (bufins(buffer,n)==ERR) {
            break;
        }
        if (bufdn()==ERR) {
            break;
        }
    }
    /* close the file */
    sysclose(file);
    /* indicate that the buffer is fresh */
    bufsaved();
    /* set current line to line 1 */
    bufgo(1);
    /* redraw the screen so that topline will be
     * on line 1 after command() does a CR/LF.
     */
    topline=max(1,bufln()-SCRNL2);
    bufout(topline,2,SCRNL2);
    bufgo(topline);
}

/* change current file name */

name(args) char *args;
```

```
{
        name1(args,filename);
        pmtfile(filename);
}

/* check syntax of args.
 * copy to filename.
 * return OK if the name is valid.
 */

name1(args,filename) char *args, *filename;
{
        /* skip command */
        args=skiparg(args);
        args=skipbl(args);
        /* check file name syntax */
        if (syschkfn(args)==ERR) {
                return(ERR);
        }
        /* copy filename */
        syscopfn(args,filename);
        return(OK);
}
```

ed3.c September 8, 1981 page 6

```
/* save the buffer in an already existing file */

resave()
{
char linebuf[MAXLEN];
int file, n, oldline;
        /* make sure file has a name */
        if (filename[0]==EOS) {
                message("file not named");
                return;
        }
        /* the file must exist for resave */
        if ((file=sysopen(filename,"r"))==ERR) {
                message("file not found");
                return;
        }
        if (sysclose(file)==ERR) {
                return;
        }
        /* open the file for writing */
        if ((file=sysopen(filename,"w"))==ERR) {
                return;
        }
        /* save the current position of file */
        oldline=bufln();
        /* write out the whole file */
        if (bufgo(1)==ERR) {
                sysclose(file);
                return;
        }
        while (bufatbot()==NO) {
                n=bufgetln(linebuf,MAXLEN);
                n=min(n,MAXLEN);
                if (pushline(file,linebuf,n)==ERR) {
                        break;
                }
                if (bufdn()==ERR) {
                        break;
                }
        }
        /* indicate if all buffer was saved */
        if (bufatbot()){
                bufsaved();
        }
        /* close file and restore line number */
        sysclose(file);
        bufgo(oldline);
}

/* save the buffer in a new file */

save()
{
char linebuf[MAXLEN];
int file, n, oldline;
        /* make sure the file is named */
        if (filename[0]==EOS) {
                message("file not named");
                return;
        }
        /* file must NOT exist for save */
        if ((file=sysopen(filename,"r"))!=ERR) {
```

ed3.c September 8, 1981 page 7

```
                sysclose(file);
                message("file exists");
                return;
        }
        /* open file for writing */
        if ((file=sysopen(filename,"w"))==ERR) {
                return;
        }
        /* remember current line */
        oldline=bufln();
        /* write entire buffer to file */
        if (bufgo(1)==ERR) {
                sysclose(file);
                return;
        }
        while (bufatbot()==NO) {
                n=bufgetln(linebuf,MAXLEN);
                n=min(n,MAXLEN);
                if (pushline(file,linebuf,n)==ERR) {
                        break;
                }
                if (bufdn()==ERR) {
                        break;
                }
        }
        /* indicate buffer saved if good write */
        if (bufatbot()) {
                bufsaved();
        }
        /* restore line and close file */
        bufgo(oldline);
        sysclose(file);
}

/* global search command */

search(args) char *args;
{
int from, to;

        if (get2args(args,&from,&to)==ERR) {
                return;
        }
        search1(from, to, NO);
}

/* search lines for a pattern.
 * if flag == YES: stop at the first match.
 *                 return -1 if no match.
 *                 otherwise return column number of match
 * if flag == NO:  print all matches found.
 */

search1(from, to, flag) int from, to, flag;
{
char pat  [MAXLEN1];            /* reserve space for EOS */
char line [MAXLEN1];
int col, n;
        /* get search mask into pat */
        fmtsout("search mask ?  ",0);
        getcmnd(pat,15);
        fmtcrlf();
```

ed3.c September 8, 1981 page 8

```
        if (pat[0]==EOS) {
                return;
        }
        /* search all lines between from and to for pat */
        while (from<=to) {
                if (chkkey()==YES) {
                        break;
                }
                if (bufgo(from++)==ERR) {
                        break;
                }
                if (bufatbot()==YES) {
                        break;
                }
                n=bufgetln(line,MAXLEN);
                n=min(n,MAXLEN);
                line[n]=EOS;
                /* ^ anchors search */
                if (pat[0]=='^') {
                        if (amatch(line,pat+1,0)==YES) {
                                if (flag==NO) {
                                        fmtcrlf();
                                        putdec(bufln(),5);
                                        fmtsout(line,5);
                                        outdeol();
                                }
                                else {
                                        return(0);
```

(Continued on next page)

Screen Oriented Editor

(Listing continued, text begins on page 18)

```
                        }
                }
                continue;
        }
        /* search whole line for match */
        col=0;
        while (col<n) {
                if (amatch(line,pat,col++)==YES) {
                        if (flag==NO) {
                                fmtcrlf();
                                putdec(bufln(),5);
                                fmtsout(line,5);
                                outdeol();
                                break;
                        }
                        else {
                                return(col-1);
                        }
                }
        }
        /* all searching is finished */
        if (flag==YES) {
                return(-1);
        }
        else {
                fmtcrlf();
        }
}

/* set tab stops for fmt routines */

tabs(args) char *args;
{
```



```
int n, junk;
        if (get2args(args,&n,&junk)==ERR) {
                return;
        }
        fmtset(n);
}

/* return YES if buffer may be drastically changed */

chkbuf()
{
        if (bufchng()==NO) {
                /* buffer not changed. no problem */
                return(YES);
        }
        fmtsout("buffer not saved. proceed ?  ",0);
        pmtline();
        if (tolower(syscout(syscin()))!='y') {
                fmtcrlf();
                message("cancelled");
                return(NO);
        }
        else {
                fmtcrlf();
                return(YES);
        }
}

/* print message from a command */

message(s) char *s;
{
        fmtsout(s,0);
        fmtcrlf();
}

/* get two arguments the argument line args.
 * no arguments imply 1 HUGE.
 * one argument implies both args the same.
 */

get2args(args,val1,val2) char *args; int *val1, *val2;
{
        /* skip over the command */
        args=skiparg(args);
        args=skipbl(args);
        if (*args==EOS) {
                *val1=1;
                *val2=HUGE;
                return(OK);
        }
        /* check first argument */
        if (number(args,val1)==NO) {
                message("bad argument");
                return(ERR);
        }
        /* skip over first argument */
        args=skiparg(args);
        args=skipbl(args);
        /* 1 arg: arg 2 is HUGE */
        if (*args==EOS) {
                *val2=HUGE;
```



```
                return(OK);
        }
        /* check second argument */
        if (number(args,val2)==NO) {
                message("bad argument");
                return(ERR);
        }
        else {
                return(OK);
        }
}

/* skip over all except EOS, and blanks */

skiparg(args) char *args;
{
        while ((*args!=EOS)&(*args!=' ')) {
                args++;
        }
        return(args);
}

/* skip over all blanks */

skipbl(args) char *args;
{
        while (*args==' ') {
                args++;
        }
        return(args);
}

/* return YES if the user has pressed any key.
 * blanks cause a transparent pause.
 */
chkkey()
{
int c;
        c=syscstat();
        if (c==0) {
                /* no character at keyboard */
                return(NO);
        }
        else if (c==' ') {
                /* pause.  another blank ends pause */
                pmtline();
                if (syscin()==' ') {
                        return(NO);
                }
        }
        /* we got a nonblank character */
        return(YES);
}

/* anchored search for pattern in text line at column col.
 * return YES if the pattern starts at col.
 */

amatch(line,pat,col) char *line, *pat; int col;
{
int k;
        k=0;
```



```
        while (pat[k]!=EOS) {
                if (pat[k]==line[col]) {
                        k++;
                        col++;
                }
                else if ((pat[k]=='?')&(line[col]!=EOS)) {
                        /* question mark matches any char */
                        k++;
                        col++;
                }
                else {
                        return(NO);
                }
        }
        /* the entire pattern matches */
        return(YES);
```

```
}

/* replace oldpat in oldline by newpat starting at col.
 * put result in newline.
 * return number of characters in newline.
 */

replace(oldline,newline,oldpat,newpat,col)
char *oldline, *newline, *oldpat, *newpat; int col;
{
int k;
char *tail, *pat;
        /* copy oldline preceding col to newline */
        k=0;
        while (k<col) {
                newline[k++]=*oldline++;
        }
        /* remember where end of oldpat in oldline is */
        tail=oldline;
        pat=oldpat;
        while (*pat++!=EOS) {
                tail++;
        }
        /* copy newpat to newline.
         * use oldline and oldpat to resolve question marks
         * in newpat.
         */
        while (*newpat!=EOS) {
                if (k>MAXLEN-1) {
                        message("new line too long");
                        return(ERR);
                }
                if (*newpat!='?') {
                        /* copy newpat to newline */
                        newline[k++]=*newpat++;
                        continue;
                }
                /* scan for '?' in oldpat */
                while (*oldpat!='?') {
                        if (*oldpat==EOS) {
                                message(
                                "too many ?'s in change mask"
                                );
                                return(ERR);
                        }
                        oldpat++;
```



```
                        oldline++;
                }
                /* copy char from oldline to newline */
                newline[k++]=*oldline++;
                oldpat++;
                newpat++;
        }
        /* copy oldline after oldpat to newline */
        while (*tail!=EOS) {
                if (k>=MAXLEN-1) {
                        message("new line too long");
                        return(ERR);
                }
                newline[k++]=*tail++;
        }
        newline[k]=EOS;
        return(k);
}
```

```
/*
 * Screen editor:  window module
 *
 * Source:  ed4.c
 * Version: August 20, 1981.
 */

/* data global to this module */

char    editbuf[MAXLEN];        /* the edit buffer */
int     editp;                  /* cursor: buffer index */
int     editpmax;               /* length of buffer */
int     edcflag;                /* buffer change flag */

/* abort any changes made to current line */

edabt()
{
        /* get unchanged line and reset cursor */
        edgetln();
        edredraw();
        edbegin();
        edcflag=NO;
}

/* put cursor at beginning of current line */

edbegin()
{
        editp=0;
        outxy(0,outgety());
}

/* change editbuf[editp] to c
 * don't make change if line would become to long
 */

edchng(c) char c;
{
char oldc;
int k;
        /* if at right margin then insert char */
        if (editp>=editpmax) {
                edins(c);
                return;
        }
        /* change char and print length of line */
        oldc=editbuf[editp];
        editbuf[editp]=c;
        fmtadj(editbuf,editp,editpmax);
        k=fmtlen(editbuf,editpmax);
        if (k>SCRNW1) {
                /* line would become too long */
                /* undo the change */
                editbuf[editp]=oldc;
                fmtadj(editbuf,editp,editpmax);
        }
        else {
                /* set change flag, redraw line */
                edcflag=YES;
                editp++;
                edredraw();
```



```
        }
}

/* delete the char to left of cursor if it exists */

eddel()
{
int k;
        /* just move left one column if past end of line */
        if (edxpos() < outgetx()) {
                outxy(outgetx()-1, outgety());
                return;
        }
        /* do nothing if cursor is at left margin */
        if (editp==0) {
                return;
        }
        edcflag=YES;
        /* compress buffer (delete char) */
        k=editp;
        while (k<editpmax) {
                editbuf[k-1]=editbuf[k];
                k++;
        }
        /* update pointers, redraw line */
        editp--;
        editpmax--;
        edredraw();
}

/* edit the next line.  do not go to end of buffer */

eddn()
{
int oldx;
        /* save visual position of cursor */
        oldx=outgetx();
        /* replace current edit line */
        if (edrepl()!=OK) {
                return(ERR);
        }
        /* do not go past last non-null line */
        if (bufnrbot()) {
                return(OK);
        }
        /* move down one line in buffer */
        if (bufdn()!=OK) {
```

(Continued on next page)

Screen Oriented Editor

(Listing continued, text begins on page 18)

```
                return(ERR);
        }
        edgetln();
        /* put cursor as close as possible on this
         * new line to where it was on the old line.
         */
        editp=edscan(oldx);
        /* update screen */
        if (edatbot()) {
                edsup(bufln()-SCRNL2);
                outxy(oldx, SCRNL1);
        }
        else {
                outxy(oldx, outgety()+1);
        }
```



```
        return(OK);
}

/* put cursor at the end of the current line */

edend()
{
        editp=editpmax;
        outxy(edxpos(),outgety());

        /* comment out ----- put cursor at end of screen
        outxy(SCRNW1, outgety());
        ----- end comment out */
}

/* start editing line n
 * redraw the screen with cursor at position p
 */

edgo(n, p) int n, p;
{
        /* replace current line */
        if (edrepl()==ERR) {
                return(ERR);
        }
        /* go to new line */
        if (bufgo(n)==ERR) {
                return(ERR);
        }
        /* prevent going past end of buffer */
        if (bufatbot()) {
                if (bufup()==ERR) {
                        return(ERR);
                }
        }
        /* redraw the screen */
        bufout(bufln(),1,SCRNL1);
        edgetln();
        editp=min(p, editpmax);
        outxy(edxpos(), 1);
        return(OK);
}

/* insert c into the buffer if possible */

edins(c) char c;
{
int k;

        /* do nothing if edit buffer is full */
        if (editpmax>=MAXLEN) {
                return;
        }
        /* fill out line if we are past its end */
        if ((editp == editpmax) & (edxpos() < outgetx())) {
                k = outgetx() - edxpos();
                editpmax = editpmax + k;
                while (k-- > 0) {
                        editbuf [editp++] = ' ';
                }
                editp = editpmax;
        }
```



```
        /* make room for inserted character */
        k=editpmax;
        while (k>editp) {
                editbuf[k]=editbuf[k-1];
                k--;
        }
        /* insert character. update pointers */
        editbuf[editp]=c;
        editp++;
        editpmax++;
        /* recalculate print length of line  */
        fmtadj(editbuf,editp-1,editpmax);
        k=fmtlen(editbuf,editpmax);
        if (k>SCRNW1) {
                /* line would become too long */
                /* delete what we just inserted */
                eddel();
        }
        else {
                /* set change flag, redraw line */
                edcflag=YES;
                edredraw();
        }
}

/* join (concatenate) the current line with the one above it */

edjoin()
{
int k;

        /* do nothing if at top of file */
        if (bufattop()) {
                return;
        }
        /* replace lower line temporarily */
        if (edrepl() != OK) {
                return;
        }
        /* get upper line into buffer */
        if (bufup() != OK) {
                return;
        }
        k = bufgetln(editbuf, MAXLEN);
        /* append lower line to buffer */
        if (bufdn() != OK) {
                return;
        }
        k = k + bufgetln(editbuf+k, MAXLEN-k);
        /* abort if the screen isn't wide enough */
        if (k>SCRNW1) {
                return;
        }
        /* replace upper line */
        if (bufup() != OK) {
                return;
        }
        editpmax = k;
        edcflag = YES;
        if (edrepl() != OK) {
                return;
        }
```



```
        /* delete the lower line */
        if (bufdn() != OK) {
                return;
        }
        if (bufdel() != OK) {
                return;
        }
        if (bufup() != OK) {
                return;
        }
        /* update the screen */
        if (edattop()) {
                edredraw();
        }
        else {
                k = outgety() - 1;
                bufout(bufln(),k,SCRNL-k);
                outxy(0,k);
                edredraw();
        }
}

/* delete chars until end of line or c found */

edkill(c) char c;
{
int k,p;
        /* do nothing if at right margin */
        if (editp==editpmax) {
                return;
        }
        edcflag=YES;
        /* count number of deleted chars */
        k=1;
```

```
        while ((editp+k)<editpmax) {
                if (editbuf[editp+k]==c) {
                        break;
                }
                else {
                        k++;
                }
        }
        /* compress buffer (delete chars) */
        p=editp+k;
        while (p<editpmax) {
                editbuf[p-k]=editbuf[p];
                p++;
        }
        /* update buffer size, redraw line */
        editpmax=editpmax-k;
        edredraw();
}

/* move cursor left one column.
 * never move the cursor off the current line.
 */

edleft()
{
int k;

        /* if past right margin, move left one column */
```



```
        if (edxpos() < outgetx()) {
                outxy(max(0, outgetx()-1), outgety());
        }
        /* inside the line.  move left one character */
        else if (editp!=0) {
                editp--;
                outxy(edxpos(),outgety());
        }
}

/* insert a new blank line below the current line */

ednewdn()
{
int k;
        /* make sure there is a current line and
         * put the current line back into the buffer.
         */
        if (bufatbot()) {
                if (bufins(editbuf,editpmax)!=OK) {
                        return;
                }
        }
        else if (edrepl()!=OK) {
                        return;
        }
        /* move past current line */
        if (bufdn()!=OK) {
                return;
        }
        /* insert place holder:  zero length line */
        if (bufins(editbuf,0)!=OK) {
                return;
        }
        /* start editing the zero length line */
        edgetln();
        /* update the screen */
        if (edatbot()) {
                /* note: bufln() >= SCRNL */
                edsup(bufln()-SCRNL2);
                outxy(edxpos(),SCRNL1);
        }
        else {
                k=outgety();
                bufout(bufln(),k+1,SCRNL1-k);
                outxy(edxpos(),k+1);
        }
}

/* insert a new blank line above the current line */

ednewup()
{
int k;
        /* put current line back in buffer */
        if (edrepl()!=OK) {
                return;
        }
        /* insert zero length line at current line */
        if (bufins(editbuf,0)!=OK) {
                return;
        }
```



```
        /* start editing the zero length line */
        edgetln();
        /* update the screen */
        if (edattop()) {
                edsdn(bufln());
                outxy(edxpos(),1);
        }
        else {
                k=outgety();
                bufout(bufln(),k,SCRNL-k);
                outxy(edxpos(),k);
        }
}

/* move cursor right one character.
 * never move the cursor off the current line.
 */

edright()
{
        /* if we are outside the line move right one column */
        if (edxpos() < outgetx()) {
                outxy (min(SCRNW1, outgetx()+1), outgety());
        }
        /* if we are inside a tab move to the end of it */
        else if (edxpos() > outgetx()) {
                outxy (edxpos(), outgety());
        }
        /* move right one character if inside line */
        else if (editp < editpmax) {
                editp++;
                outxy(edxpos(),outgety());
        }
        /* else move past end of line */
        else {
                outxy (min(SCRNW1, outgetx()+1), outgety());
        }
}

/* split the current line into two parts.
 * scroll the first half of the old line up.
 */

edsplit()
{
int p, q;
int k;

        /* indicate that edit buffer has been saved */
        edcflag = NO;
        /* replace current line by the first half of line */
        if (bufatbot()) {
                if (bufins(editbuf, editp) != OK) {
                        return;
                }
        }
        else {
                if (bufrepl(editbuf, editp) != OK) {
                        return;
                }
        }
        /* redraw the first half of the line */
```



```
        p = editpmax;
        q = editp;
        editpmax = editp;
        editp = 0;
        edredraw();
        /* move the second half of the line down */
        editp = 0;
        while (q < p) {
                editbuf [editp++] = editbuf [q++];
        }
        editpmax = editp;
        editp = 0;
        /* insert second half of the line below the first */
        if (bufdn() != OK) {
                return;
        }
        if (bufins(editbuf, editpmax) != OK) {
                return;
        }
        /* scroll the screen up and draw the second half */
        if (edatbot()) {
                edsup(bufln()-SCRNL2);
                outxy(1,SCRNL1);
                edredraw();
        }
```

(Continued on next page)

Screen Oriented Editor

(Listing continued, text begins on page 18)

```
        else {
                k = outgety();
                bufout(bufln(), k+1, SCRNL1-k);
                outxy(1, k+1);
                edredraw();
        }
}

/* move cursor right until end of line or
 * character c found.
 */

edsrch(c) char c;
{
        /* do nothing if at right margin */
        if (editp==editpmax) {
                return;
        }
        /* scan for search character */
        editp++;
        while (editp<editpmax) {
                if (editbuf[editp]==c) {
                        break;
                }
                else {
                        editp++;
                }
        }
        /* reset cursor */
        outxy(edxpos(),outgety());
}

/* move cursor up one line if possible */

edup()
{
int oldx;
```



```
        /* save visual position of cursor */
        oldx=outgetx();
        /* put current line back in buffer */
        if (edrepl()!=OK) {
                return(ERR);
        }
        /* done if at top of buffer */
        if (bufattop()) {
                return(OK);
        }
        /* start editing the previous line */
        if (bufup()!=OK) {
                return(ERR);
        }
        edgetln();
        /* put cursor on this new line as close as
         * possible to where it was on the old line.
         */
        editp=edscan(oldx);
        /* update screen */
        if (edattop()) {
                edsdn(bufln());
                outxy(oldx, 1);
        }
        else {
                outxy(oldx, outgety()-1);
        }
        return(OK);
}

/* delete the current line */

edzap()
{
int k;
        /* delete the line in the buffer */
        if (bufdel()!=OK) {
                return;
        }
        /* move up one line if now at bottom */
        if (bufatbot()) {
                if (bufup()!=OK) {
                        return;
                }
                edgetln();
                /* update screen */
                if (edattop()) {
                        edredraw();
                }
                else {
                        outdelln();
                        outxy(0,outgety()-1);
                }
                return;
        }
        /* start editing new line */
        edgetln();
        /* update screen */
        if (edattop()) {
                edsup(bufln());
                outxy(0,1);
        }
```



```
        else {
                k=outgety();
                bufout(bufln(),k,SCRNL-k);
                outxy(0,k);
        }
}

/* return true if the current edit line is being
 * displayed on the bottom line of the screen.
 */

edatbot()
{
        return(outgety()==SCRNL1);
}

/* return true if the current edit line is being
 * displayed on the bottom line of the screen.
 */

edattop()
{
        return(outgety()==1);
}

/* redraw edit line from index to end of line */
/* reposition cursor */

edredraw()
{
        fmtadj(editbuf,0,editpmax);
        fmtsubs(editbuf,max(0,editp-1),editpmax);
        outxy(edxpos(),outgety());
}

/* return the x position of the cursor on screen */

edxpos()
{
        return(min(SCRNW1,fmtlen(editbuf,editp)));
}


/* fill edit buffer from current main buffer line.
 * the caller must check to make sure the main
 * buffer is available.
 */

edgetln()
{
int k;
        /* put cursor on left margin, reset flag */
        editp=0;
        edcflag=NO;
        /* get edit line from main buffer */
        k=bufgetln(editbuf,MAXLEN);
        if (k>MAXLEN) {
                error("line truncated");
                editpmax=MAXLEN;
        }
        else {
                editpmax=k;
```



```
        }
        fmtadj(editbuf,0,editpmax);
}

/* replace current main buffer line by edit buffer.
 * the edit buffer is NOT changed or cleared.
 * return ERR is something goes wrong.
 */

edrepl()
{
        /* do nothing if nothing has changed */
        if (edcflag==NO) {
                return(OK);
```

```
        }
        /* make sure we don't replace the line twice */
        edcflag=NO;
        /* insert instead of replace if at bottom of file */
        if (bufatbot()) {
                return(bufins(editbuf,editpmax));
        }
        else {
                return(bufrepl(editbuf,editpmax));
        }
}

/* set editp to the largest index such that
 * buf[editp] will be printed <= xpos
 */

edscan(xpos) int xpos;
{
        editp=0;
        while (editp<editpmax) {
                if (fmtlen(editbuf,editp)<xpos) {
                        editp++;
                }
                else {
                        break;
                }
        }
        return(editp);
}

/* scroll the screen up.  topline will be new top line */

edsup(topline) int topline;
{
        if (outhasup()==YES) {
                /* hardware scroll */
                outsup();
                /* redraw bottom line */
                bufout(topline+SCRNL2,SCRNL1,1);
        }
        else {
                /* redraw whole screen */
                bufout(topline,1,SCRNL1);
        }
}

/* scroll screen down.  topline will be new top line */
```



```
edsdn(topline) int topline;
{
        if (outhasdn()==YES) {
                /* hardware scroll */
                outsdn();
                /* redraw top line */
                bufout(topline,1,1);
        }
        else {
                /* redraw whole screen */
                bufout(topline,1,SCRNL1);
        }
}
```

```
/* Screen editor:  output format module
 *
 * Source: ed5.c
 * Version: March 6, 1981.
 */


/* Define variables global to this module */

/* define maximal length of a tab character */

int fmttab;

/* define the current device and device width */

int fmtdev;             /* device -- YES/NO = LIST/CONSOLE */
int fmtwidth;           /* devide width.  LISTW/SCRNW1 */

/* fmtcol[i] is the first column at which
 * buf[i] will be printed.
 * fmtsub() and fmtlen() assume fmtcol[] is valid on entry.
 */

int fmtcol[MAXLEN1];

/* direct output from this module to either the console or
 * the list device.
 */

fmtassn(listflag) int listflag;
{
        if (listflag==YES) {
                fmtdev=YES;
                fmtwidth=LISTW;
        }
        else {
                fmtdev=NO;
                fmtwidth=SCRNW1;
        }
}

/* adjust fmtcol[] to prepare for calls on
 * fmtout() and fmtlen()
 *
 * NOTE:  this routine is needed as an efficiency
 *        measure.  Without fmtadj(), calls on
 *        fmtlen() become too slow.
 */

fmtadj(buf,minind,maxind) char *buf; int minind,maxind;
{
int k;
        /* line always starts at left margin */
        fmtcol[0]=0;
        /* start scanning at minind */
        k=minind;
        while (k<maxind) {
                if (buf[k]==CR) {
                        break;
                }
                fmtcol[k+1]=fmtcol[k]+fmtlench(buf[k],fmtcol[k]);
                k++;
```



```
        }
}

/* return column at which at which buf[i] will be printed */

fmtlen(buf,i) char *buf; int i;
{
        return(fmtcol[i]);
}

/* print buf[i] ... buf[j-1] on current device so long as
 * characters will not be printed in last column.
 */

fmtsubs(buf,i,j) char *buf; int i, j;
{
int k;
        if (fmtcol[i]>=fmtwidth) {
                return;
        }
        outxy(fmtcol[i],outgety());     /* position cursor */
        while (i<j) {
                if (buf[i]==CR) {
                        break;
                }
                if (fmtcol[i+1]>fmtwidth) {
                        break;
                }
                fmtoutch(buf[i],fmtcol[i]);
                i++;
        }
        outdeol();      /* clear rest of line */
}

/* print string which ends with CR or EOS to current device.
 * truncate the string if it is too long.
 */

fmtsout(buf,offset) char *buf; int offset;
{
char c;
int col,k;
        col=0;
        while (c=*buf++) {
                if (c==CR) {
                        break;
                }
                k=fmtlench(c,col);
                if ((col+k+offset)>fmtwidth) {
                        break;
                }
                fmtoutch(c,col);
                col=col+k;
        }
```

(Continued on next page)

Screen Oriented Editor

(Listing continued, text begins on page 18)

```
}

/* return length of char c at column col */

fmtlench(c,col) char c; int col;
{
        if (c==TAB) {
                /* tab every fmttab columns */
```



```
                return(fmttab-(col%fmttab));
        }
        else if (c<32) {
                /* control char */
                return(2);
        }
        else {
                return(1);
        }
}

/* output one character to current device.
 * convert tabs to blanks.
 */

fmtoutch(c,col) char c; int col;
{
int k;
        if (c==TAB) {
                k=fmtlench(TAB,col);
                while ((k--)>0) {
                        fmtdevch(' ');
                }
        }
        else if (c<32) {
                fmtdevch('^');
                fmtdevch(c+64);
        }
        else {
                fmtdevch(c);
        }
}

/* output character to current device */

fmtdevch(c) char c;
{
        if (fmtdev==YES) {
                syslout(c);
        }
        else {
                outchar(c);
        }
}

/* output a CR and LF to the current device */

fmtcrlf()
{
        if (fmtdev==YES) {
                syslout(CR);
                syslout(LF);
        }
        else {
                /* kludge: this should be in out module */
                /* make sure out module knows position */
                outxy(0,SCRNL1);
                syscout(CR);
                syscout(LF);
        }
}
```



```
/* set tabs at every n columns */

fmtset(n) int n;
{
        fmttab=max(1,n);
}
```

```
/*
Screen editor:  terminal output module
Source:  ed6.ccc
This file was created by the configuration program:
Version 2:  September 6, 1981.
*/

/*
Define the current coordinates of the cursor.
*/

int outx, outy;

/*
Return the current coordinates of the cursor.
*/

outgetx()
{
        return(outx);
}

outgety()
{
        return(outy);
}

/*
Output one printable character to the screen.
*/

outchar(c) char c;
{
        syscout(c);
        outx++;
        return(c);
}

/*
Position cursor to position x,y on screen.
0,0 is the top left corner.
*/

outxy(x,y) int x,y;
{
        outx=x;
        outy=y;
        syscout(27);
        syscout('=');
        syscout(x+32);
        syscout(y+32);
}

/*
Erase the entire screen.
Make sure the rightmost column is erased.
*/

outclr()
{
int k;
```



```
        k=0;
        while (k<SCRNL) {
                outxy(0,k++);
                outdelln();
        }
        outxy(0,0);
}

/*
Delete the line on which the cursor rests.
Leave the cursor at the left margin.
*/

outdelln()
{
        outxy(0,outy);
        outdeol();
}

/*
Delete to end of line.
Assume the last column is blank.
*/

outdeol()
{
        syscout(27);
        syscout('E');
}

/*
```

```
Return yes if terminal has indicated hardware scroll.
*/

outhasup()
{
        return(YES);
}

outhasdn()
{
        return(YES);
}

/*
Scroll the screen up.
Assume the cursor is on the bottom line.
*/

outsup()
{
        /* auto scroll */
        outxy(0,SCRNL1);
        syscout(10);
}

/*
Scroll screen down.
Assume the cursor is on the top line.
*/

outsdn()
```



```
{
        /* auto scroll */
        outxy(0,0);
        syscout(27);
        syscout('^');
}
```



```
/* Screen editor:  prompt line module
 *
 * Source:  ed7.c
 * Version: March 6, 1981.
 */

/* Define the prompt line data */

char pmtln[MAXLEN];     /* mode */
char pmtfn[SYSFNMAX];   /* file name */


/* put error message on prompt line.
 * wait for response.
 */

pmtmess(s1,s2) char *s1, *s2;
{
int x,y;
        /* save cursor */
        x=outgetx();
        y=outgety();
        outxy(0,0);
        /* make sure line is correct */
        outdelln();
        pmtline1();
        pmtcol1(x);
        /* output error message */
        fmtsout(s1,outgetx());
        fmtsout(s2,outgetx());
        /* wait for input from console */
        syscin();
        /* redraw prompt line */
        pmtline1();
        pmtcol1(x);
        pmtfile1(pmtfn);
        pmtmode1(pmtln);
        /* restore cursor */
        outxy(x,y);
}

/* write new mode message on prompt line */

pmtmode(s) char *s;
{
int x,y;                /* save cursor on entry */
        /* save cursor */
        x=outgetx();
        y=outgety();
        /* redraw whole line */
        outxy(0,0);
        outdelln();
        pmtline1();
        pmtcol1(x);
        pmtfile1(pmtfn);
        pmtmode1(s);
        /* restore cursor */
        outxy(x,y);
}

/* update file name on prompt line */
```



```
pmtfile(s) char *s;
{
int x, y;
        /* save cursor */
        x=outgetx();
        y=outgety();
        /* update whole line */
        outxy(0,0);
        outdelln();
        pmtline1();
        pmtcol1();
        pmtfile1(s);
        pmtmode1(pmtln);
        /* restore cursor */
        outxy(x,y);
}

/* change mode on prompt line to edit: */

pmtedit()
{
        pmtmode("edit:");
}

/* update line and column numbers on prompt line */

pmtline()
{
int x,y;
        /* save cursor */
        x=outgetx();
        y=outgety();
        /* redraw whole line */
        outxy(0,0);
        outdelln();
        pmtline1();
        pmtcol1(x);
        pmtfile1(pmtfn);
        pmtmode1(pmtln);
        /* restore cursor */
        outxy(x,y);
}

/* update just the column number on prompt line */

pmtcol()
{
int x,y;
        /* save cursor */
        x=outgetx();
        y=outgety();
        /* update column number */
        pmtcol1(x);
        /* update cursor */
        outxy(x,y);
}

/* update mode.  call getcmnd() to write on prompt line */

pmtcmnd(mode,buffer) char *mode, *buffer;
{
int x,y;
```



```
        /* save cursor */
        x=outgetx();
        y=outgety();
        pmtmode1(mode);
        /* user types command on prompt line */
        getcmnd(buffer,outgetx());
        /* restore cursor */
}

/* update and print mode */

pmtmode1(s) char *s;
{
int i;
```

(Continued on next page)

Screen Oriented Editor

(Listing continued, text begins on page 18)

```
        outxy(40,0);
        fmtsout(s,40);
        i=0;
        while (pmtln[i++]=*s++) {
                ;
        }
}

/* print the file name on the prompt line */

pmtfile1(s) char *s;
{
int i;
        outxy(25,0);
        if (*s==EOS) {
                fmtsout("no file",25);
        }
        else {
                fmtsout(s,25);
        }
        i=0;
        while (pmtfn[i++]=*s++) {
                ;
        }
}

/* print the line number on the prompt line */

pmtline1()
{
        outxy(0,0);
        fmtsout("line: ",0);
        putdec(bufln(),5);
}


/* print column number of the cursor */

pmtcol1(x) int x;
{
        outxy(12,0);
        fmtsout("column: ",12);
        putdec(x,3);
}
```

```
ed8.c                September 8, 1981                page 1

/* Screen editor:  operating system module
 *
 * Source:  ed8.c
 * Version: June 19, 1981.
 */

/* all calls to the operating system are made here.
 * only this module and the assembler libraries will have to
 * rewritten for a new operating system.
 */

/* the routines syscstat(), syscin() and syscout() come in
 * two flavors:  CP/M version 2.2 and CP/M version 1.4.
 * Comment out which ever you don't use.
 */

/* CP/M 2.2 versions of syscstat(), syscin(), syscout() */

/* return -1 if no character is ready from the console.
 * otherwise, return the character.
 */

syscstat()
{
        return(cpm(6,-1));
}

/* wait for next character from the console.
 * do not echo it.
 */

syscin()
{
int c;
        while ((c=cpm(6,-1))==0) {
                ;
        }
        return(c);
}

/* print character on the console */

syscout(c) char c;
{
        cpm(6,c);
        return(c);
}

/* CP/M 1.4 versions of syscstat(), syscin(), syscout() */

/* start comment out ----------
#asm
;
;       bios(n,bc)
;
;       call the n'th bios routine.
;       bc is put into the bc-register as a parameter.
;       returns whatever bios puts into the a-reg.
;
QZBIOS:
        POP     H       ;get return address
        POP     B       ;    bc
ed8.c                September 8, 1981                page 2

        POP     D       ;    n
        PUSH    D       ;restore stack the way it was
        PUSH    B
        PUSH    H
        LHLD    1       ;point hl at start of BIOS jump table
        DCX     H
        DCX     H
        DCX     H
        MOV     A,E     ;put index into jump table into de
        ADD     A
        ADD     E
        MVI     D,0
        MOV     E,A
        DAD     D       ;point hl at proper entry in jump table
        PUSH    H       ;push call address
        LXI     H,QZBIOS1 ;point hl at return address
        XTHL            ;push return address, get call address
        PCHL            ;call proper BIOS routine
QZBIOS1:
        MOV     L,A     ;put return code from BIOS into hl
        MVI     H,0
        RET             ;go back to small-c
#endasm

syscstat()
{
        if (bios(2,0)==255) {
                return(-1);
        }
        else {
                return(0);
        }
}

syscin()
{
        return(bios(3,0));
}

syscout(c) char c;
{
        bios(4,c);
        return(c);
}
---------- end comment out */


/* print character on the printer */

syslout(c) char c;
{
        cpm(5,c);
        return(c);
}

#asm
;
; sysend()
;
; return address of last usable memory location.
;
QZSYSEND:
ed8.c                September 8, 1981                page 3

        LHLD    6       ;get address of BDOS-1
        DCX     H       ;
        RET             ;return
#endasm

/* open a file */

sysopen(name,mode) char *name, *mode;
```

```
{
int file;
        if ((file=fopen(name,mode))==0) {
                return(ERR);
        }
        else {
                return(file);
        }
}

/* close a file */

sysclose(file) int file;
{
        /* fclose doesn't reliably return OK */
        fclose(file);
        return(OK);
}

/* read next char from file */

sysrdch(file) int file;
{
int c;
        if ((c=getc(file))==-1) {
                return(EOF);
        }
        else {
                return(c);
        }
}

/* write next char to file */

syspshch(c,file) char c; int file;
{
        if (putc(c,file)==-1) {
                error("disk write failed");
                return(ERR);
        }
        else {
                return(c);
        }
}

/* read one char from END of file */

syspopch(file) int file;
{
        error("syspopch() not implemented");
        return(ERR);
}

/* check file name for syntax */
```



```
syschkfn(args) char *args;
{
        return(OK);
}

/* copy file name from args to buffer */

syscopfn(args,buffer) char *args, *buffer;
{
int n;
        n=0;
        while (n< SYSFNMAX-1) {
                if (args[n]==EOS) {
                        break;
                }
                else {
                        buffer[n]=args[n];
                        n++;
                }
        }
        buffer[n]=EOS;
}

/* move a block of n bytes down (towards HIGH addresses).
 * block starts a source and the first byte goes to dest.
 * this routine is only called from bufmovdn() as follows
 *      sysmovdn( n=to-from+1, dest=to+length, source=to);
 */

#asm
QZSYSMOVDN:
        POP     H       ;get return address
        POP     B       ;BC = source
        POP     D       ;DE = dest
        XTHL            ;HL = n, restore return address
        JMP     SYSMOVDN2       ;go enter loop
;
; this code is 15 times faster than the c-code it replaces.
SYSMOVDN1:
        LDAX    B       ;*dest-- = *source--
        STAX    D
        DCX     B
        DCX     D
        DCX     H       ;while ((n--)>0)
SYSMOVDN2:
        MOV     A,L
        ORA     H
        JNZ     SYSMOVDN1
;
        XTHL            ;HL = return address
        PUSH    H       ;restore stack
        PUSH    H
        PUSH    H
        RET             ;return
#endasm

/* move a block of n bytes up (towards LOW addresses).
 * the block starts at source and the first byte goes to dest.
 * this routine is called only from bufmovup() as follows:
 *      sysmovup( n=to-from+1, dest=from-length, source=from);
 */
```



```
#asm
QZSYSMOVUP:
        POP     H       ;get return address
        POP     B       ;BC = source
        POP     D       ;DE = dest
        XTHL            ;HL = n, save return address
        JMP     SYSMOVUP2       ;go enter loop
;
; this code is 15 times faster than the c-code it replaces.
SYSMOVUP1:
        LDAX    B       ;*dest++ = *source++
        STAX    D
        INX     B
        INX     D
        DCX     H       ;while ((n--)>0)
SYSMOVUP2:
        MOV     A,L
        ORA     H
        JNZ     SYSMOVUP1
;
        XTHL            ;HL = return address
        PUSH    H       ;restore stack
        PUSH    H
        PUSH    H
        RET             ;return
#endasm
```



```
/* Screen editor:  general utilities
 *
 * Source:  ed9.c
 * Version: May 3, 1981.
 */

/* convert lower case to upper case */

toupper(c) int c;
{
        if ((c<'a')|(c>'z')) {
                return(c);
        }
        else {
                return(c-32);
        }
}

/* convert upper case to lower case */

tolower(c) int c;
{
        if ((c<'A')|(c>'Z')) {
                return(c);
        }
        else {
                return(c+32);
        }
}

/* return: is first token in args a number ? */
/* return value of number in *val            */

number(args,val) char *args; int *val;
{
char c;
        c=*args++;
```

(Continued on next page)

Screen Oriented Editor

(Listing continued, text begins on page 18)

```
        if ((c<'0')|(c>'9')) {
                return(NO);
        }
        *val=c-'0';
        while (c=*args++) {
                if ((c<'0')|(c>'9')) {
                        break;
                }
                *val=(*val*10)+c-'0';
        }
        return(YES);
}

/* convert character buffer to numeric */

ctoi(buf,index) char *buf; int index;
{
int k;
        while ( (buf[index]==' ') |
                (buf[index]==TAB) ) {
                index++;
        }
        k=0;
        while ((buf[index]>='0')&(buf[index]<='9')) {
                k=(k*10)+buf[index]-'0';
```



```
                index++;
        }
        return(k);
}


/* return maximum of m,n */

max(m,n) int m,n;
{
        if (m>=n) {
                return(m);
        }
        else {
                return(n);
        }
}

/* return minimum of m,n */

min(m,n) int m,n;
{
        if (m<=n) {
                return(m);
        }
        else {
                return(n);
        }
}

/* put decimal integer n in field width >= w.
 * left justify the number in the field.
 */

putdec(n,w) int n,w;
{
char chars[10];
int i,nd;
        nd=itoc(n,chars,10);
        i=0;
        while (i<nd) {
                syscout(chars[i++]);
        }
        i=nd;
        while (i++<w) {
                syscout(' ');
        }
}

/* convert integer n to character string in str */

itoc(n,str,size) int n; char *str; int size;
{
int absval;
int len;
int i,j,k;
        absval=abs(n);
        /* generate digits */
        str[0]=0;
        i=1;
        while (i<size) {
                str[i++]=(absval%10)+'0';
```



```
                absval=absval/10;
                if (absval==0) {
                        break;
                }
        }
        /* generate sign */
        if ((i<size)&(n<0)) {
                str[i++]='-';
        }
        len=i-1;
        /* reverse sign, digits */
        i--;
        j=0;
        while (j<i) {
                k=str[i];
                str[i]=str[j];
                str[j]=k;
                i--;
                j++;
        }
        return(len);
}

/* return absolute value of n */

abs(n) int n;
{
        if (n<0) {
                return(-n);
        }
        else {
                return(n);
        }
}

/* system error routine */

syserr(s) char *s;
{
        pmtmess("system error: ",s);
}

/* user error routine */

error(s) char *s;
{
        pmtmess("error: ",s);
}

/* disk error routine */

diskerr(s) char *s;
{
        pmtmess("disk error: ",s);
}

/* read the next line of the file into
 * the buffer of size n that p points to.
 * Successive calls to readline() read the file
 * from front to back.
 */
```



```
readline(file,p,n) int file; char *p; int n;
{
int c;
int k;
        k=0;
        while (1) {
                c=sysrdch(file);
                if (c==ERR) {
                        return(ERR);
                }
                if (c==EOF) {
                        /* ignore line without CR */
                        return (EOF);
                }
                if (c==CR) {
                        return(k);
                }
                if (k<n) {
                        /* move char to buffer */
                        *p++=c;
                }
                /* always bump count */
                k++;
        }
}

/* push (same as write) line to end of file.
```

```
 * line is in the buffer of size n that p points to.
 * lines written by this routine may be read by
 * either readline() or popline().
 */

pushline(file,p,n) int file; char *p; int n;
{
        /* write all but trailing CR */
        while ((n--)>0) {
                if (syspshch(*p++,file)==ERR) {
                        return(ERR);
                }
        }
        /* write trailing CR */
        return(syspshch(CR,file));
}

/* pop a line from the back of the file.
 * the line should have been pushed using pushline().
 */

popline(file,p,n) int file; char *p; int n;
{
int c;
int k, kmax, t;
        /* first char must be CR */
        c=syspopch(file);
        if (c==EOF) {
                /* at START of file */
                return(EOF);
        }
        if (c==CR) {
                /* put into buffer */
                *p++=CR;
                k=1;
```



```
        }
        else {
                syserr("popline: missing CR");
                return(ERR);
        }
        /* pop line into buffer in reverse order */
        while (1) {
                c=syspopch(file);
                if (c==ERR) {
                        return(ERR);
                }
                if (c==EOF) {
                        break;
                }
                if (c==CR) {
                        /* this ends ANOTHER line */
                        /* push it back           */
                        if (syspshch(CR,file)==ERR) {
                                return(ERR);
                        }
                        break;
                }
                /* non-special case */
                if (k<n) {
                        /* put into buffer */
                        *p++=c;
                }
                /* always bump count */
                k++;
        }
        /* remember if we truncated the line */
        kmax=k;
        /* reverse the buffer */
        k=min(k,n-1);
        t=0;
        while (k>t) {
                /* swap p[t], p[k] */
                c=p[k];
                p[k]=p[t];
                p[t]=c;
                k--;
                t++;
        }
        return(kmax);
}
```



```
/* Screen editor:  buffer module
 *
 * Source:  ed10.c
 * Version: April 7, 1981.
 */

/* Define the variables global to this module.
 * Buffer must be declared after all other variables
 * of the entire program.
 * Note: buffer must have nonzero dimension.
 */

int bufcflag;           /* main buffer changed flag */
char *bufp;             /* start of current line */
char *bufpmax;          /* end of last line */
char *bufend;           /* last byte of buffer */
int bufline;            /* current line number */
int bufmaxln;           /* number of lines in buffer */
char buffer[1];         /* start of buffer */

/* This code is built around several invariant
 * assumptions:
 * First, the last line is always completely empty.
 * When bufp points to the last line there is NO
 * CR following it.
 * Second, bufp points to the last line if and only if
 * bufline==bufmaxln+1.
 * Third, bufline is always greater than zero.
 * Line zero exists only to make scanning for the
 * start of line one easier.
 */


/* Clear the main buffer */

bufnew()
{
        /* point past line 0 */
        bufp=bufpmax=buffer+1;
        /* point at last byte of buffer */
        /* allow space for stack */
        bufend=sysend()-1000;
        /* at line one, no lines in buffer */
        bufline=1;
        bufmaxln=0;
        /* line zero is always a null line */
        buffer[0]=CR;
        /* indicate no need to save file yet */
        bufcflag=NO;
}

/* return current line number */

bufln()
{
        return(bufline);
}

/* return YES if the buffer (i.e., file) has been
 * changed since the last time the file was saved.
 */
```



```
bufchng()
{
        return(bufcflag);
}

/* the file has been saved.  clear bufcflag. */

bufsaved()
{
        bufcflag=NO;
}

/* return number of bytes left in the buffer */

buffree()
{
        return(bufend-bufp);
}

/* Position buffer pointers to start of indicated line */

bufgo(line) int line;
{
        /* put request into range. prevent extension */
        line=min(bufmaxln+1,line);
        line=max(1,line);
        /* already at proper line? return. */
        if (line==bufline) {
                return(OK);
        }
        /* move through buffer one line at a time */
        while (line<bufline) {
                if (bufup()==ERR) {
                        return(ERR);
                }
        }
        while (line>bufline) {
                if (bufdn()==ERR) {
                        return(ERR);
```

(Continued on next page)

Screen Oriented Editor

(Listing continued, text begins on page 18)

```
                }
        }
        /* we have reached the line we wanted */
        return(OK);
}

/* move one line closer to front of buffer, i.e.,
 * set buffer pointers to start of previous line.
 */

bufup()
{
char *oldbufp;
        oldbufp=bufp;
        /* can't move past line 1 */
        if (bufattop()) {
                return(OK);
        }
        /* move past CR of previous line */
        if (*--bufp!=CR) {
                syserr("bufup: missing CR");
                bufp=oldbufp;
                return(ERR);
```



```
        }
        /* move to start of previous line */
        while (*--bufp!=CR) {
                ;
        }
        bufp++;
        /* make sure we haven't gone too far */
        if (bufp<(buffer+1)) {
                syserr("bufup: bufp underflow");
                bufp=oldbufp;
                return(ERR);
        }
        /* success!  we ARE at previous line */
        bufline--;
        return(OK);
}

/* Move one line closer to end of buffer, i.e.,
 * set buffer pointers to start of next line.
 */

bufdn()
{
char *oldbufp;
        oldbufp=bufp;
        /* do nothing silly if at end of buffer */
        if (bufatbot()) {
                return(OK);
        }
        /* scan past current line and CR */
        while (*bufp++!=CR) {
                ;
        }
        /* make sure we haven't gone too far */
        if (bufp>bufpmax) {
                syserr("bufdn: bufp overflow");
                bufp=oldbufp;
                return(ERR);
        }
        /* success! we are at next line */
        bufline++;
        return(OK);
}

/* Insert a line before the current line.
 * p points to a line of length n to be inserted.
 * Note: n does not include trailing CR.
 */

bufins(p,n) char *p; int n;
{
int k;
        /* make room in the buffer for the line */
        if (bufext(n+1)==ERR) {
                return(ERR);
        }
        /* put the line and CR into the buffer */
        k=0;
        while (k<n) {
                *(bufp+k)=*(p+k);
                k++;
        }
```



```
        *(bufp+k)=CR;
        /* increase number of lines in buffer */
        bufmaxln++;
        /* special case: inserting a null line at
         * end of file is not a significant change.
         */
        if ((n==0)&(bufnrbot())) {
                ;
        }
        else {
                bufcflag=YES;
        }
        return(OK);
}

/* delete the current line */

bufdel()
{
        return(bufdeln(1));
}

/* delete n lines, starting with the current line */

bufdeln(n) int n;
{
int oldline;
int k;
char *oldbufp;
        /* remember current buffer parameters */
        oldline=bufline;
        oldbufp=bufp;
        /* scan for first line after deleted lines */
        k=0;
        while ((n--)>0) {
                if (bufatbot()) {
                        break;
                }
                if (bufdn()==ERR) {
                        bufline=oldline;
                        oldbufp=bufp;
                        return(ERR);
                }
                k++;
        }
        /* compress buffer.  update pointers */
        bufmovup(bufp,bufpmax-1,bufp-oldbufp);
        bufpmax=bufpmax-(bufp-oldbufp);
        bufp=oldbufp;
        bufline=oldline;
        bufmaxln=bufmaxln-k;
        bufcflag=YES;
        return(OK);
}

/* replace current line with the line that
 * p points to.  The new line is of length n.
 */

bufrepl(p,n) char *p; int n;
{
int oldlen, k;
```



```
char *nextp;
        /* do not replace null line.  just insert */
        if (bufatbot()) {
                return(bufins(p,n));
        }
        /* point nextp at start of next line */
        if (bufdn()==ERR) {
                return(ERR);
        }
        nextp=bufp;
        if (bufup()==ERR) {
                return(ERR);
        }
        /* allow for CR at end */
        n=n+1;
        /* see how to move buffer below us;
         * up, down, or not at all.
         */
        oldlen=nextp-bufp;
        if (oldlen<n) {
                /* move buffer down */
                if (bufext(n-oldlen)==ERR) {
                        return(ERR);
                }
                bufpmax=bufpmax+n-oldlen;
        }
        else if (oldlen>n) {
```

```
                /* move buffer up */
                bufmovup(nextp,bufpmax-1,oldlen-n);
                bufpmax=bufpmax-(oldlen-n);
        }
        /* put new line in the hole we just made */
        k=0;
        while (k<(n-1)) {
                bufp[k]=p[k];
                k++;
        }
        bufp[k]=CR;
        bufcflag=YES;
        return(OK);
}

/* copy current line into buffer that p points to.
 * the maximum size of that buffer is n.
 * return k=length of line in the main buffer.
 * if k>n then truncate n-k characters and only
 * return n characters in the caller's buffer.
 */

bufgetln(p,n) char *p; int n;
{
int k;
        /* last line is always null */
        if (bufatbot()) {
                return(0);
        }
        /* copy line as long as it not too long */
        k=0;
        while (k<n) {
                if (*(bufp+k)==CR) {
                        return(k);
                }
```



```
                *(p+k)=*(bufp+k);
                k++;
        }
        /* count length but move no more chars */
        while (*(bufp+k)!=CR) {
                k++;
        }
        return(k);
}

/* move buffer down (towards HIGH addresses) */

bufmovdn(from,to,length) char *from, *to; int length;
{
        /* this code needs to be very fast.
         * use an assembly language routine.
         */
        sysmovdn(to-from+1,to+length,to);
}

/* the call to sysmovdn() is equivalent to the following code:

int k;
        k=to-from+1;
        while ((k--)>0) {
                *(to+length)=*to;
                to--;
        }

 */

/* move buffer up (towards LOW addresses) */

bufmovup(from,to,length) char *from, *to; int length;
{
        /* this code must be very fast.
         * use an assembly language routine.
         */
        sysmovup(to-from+1,from-length,from);
}

/* the call to sysmovup() is equivalent to the following code:

int k;
        k=to-from+1;
        while ((k--)>0) {
                *(from-length)=*from;
                from++;
        }

 */

/* return true if at bottom of buffer.
 * NOTE 1: the last line of the buffer is always null.
 * NOTE 2: the last line number is always bufmaxln+1.
 */

bufatbot()
{
        return(bufline>bufmaxln);
}
```



```
/* return true if at bottom or at the last
 * real line before the bottom.
 */

bufnrbot()
{
        return(bufline>=bufmaxln);
}

/* return true if at top of buffer */

bufattop()
{
        return(bufline==1);
}

/* put nlines lines from buffer starting with
 * line topline at position topy of the screen.
 */

bufout(topline,topy,nlines) int topline, topy, nlines;
{
int l,p;
        /* remember buffer's state */
        l=bufline;
        p=bufp;
        /* write out one line at a time */
        while ((nlines--)>0) {
                outxy(0,topy++);
                bufoutln(topline++);
        }
        /* restore buffer's state */
        bufline=l;
        bufp=p;
}

/* print line of main buffer on screen */

bufoutln(line) int line;
{
        /* error message does NOT go on prompt line */
        if (bufgo(line)==ERR) {
                fmtsout("disk error: line deleted",0);
                outdeol();
                return;
        }
        /* blank out lines below last line of buffer */
        if (bufatbot()) {
                outdeol();
        }
        /* write one formatted line out */
        else {
                fmtsout(bufp,0);
                outdeol();
        }
}

/* simple memory version of bufext.
 * create a hole in buffer at current line.
 * length is the size of the hole.
 */
```



```
bufext(length) int length;
{
        /* make sure there is room for more */
        if ((bufpmax+length)>=bufend) {
                error("main buffer is full");
                return(ERR);
        }
        /* move lines below current line down */
        bufmovdn(bufp,bufpmax-1,length);
        bufpmax=bufpmax+length;
        return(OK);
}
```

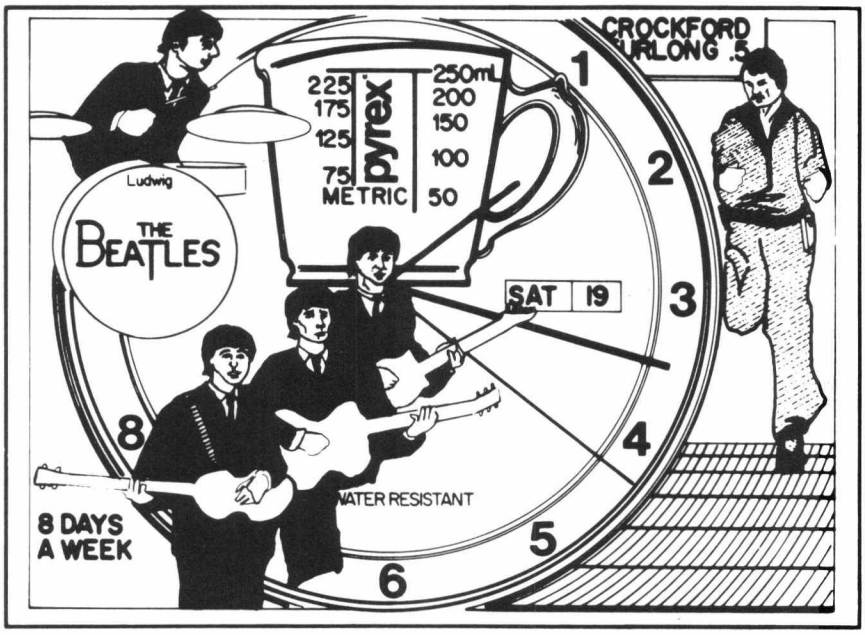
End Listing

DR. DOBB'S JOURNAL

For Users of Small Computer Systems

Number 64 February 1982 Volume 7, Issue 2

$2.50
($3.00 in Canada)



Excuse Me, What Time Is It?
Alan Bomberger

Background Information on Text Editors
Leonard K. Nicholson

Includes Special FORTH Section:

FORTH 8086 Assembler
Ray Duncan

Formal Definition of FORTH As a Programming Language
C.H. Ting

Recursion in FORTH
Glyn Emery

and much more inside

When (and How) to Use Interrupts . . .

. . . a Look at Standard Measures . . .

. . . and more on FORTH!

DR. DOBB'S JOURNAL
For Users of Small Computer Systems

February 1982 Volume 7, Issue 2, Number 64

CONTENTS

Recursion in FORTH 12
by Glyn Emery

FORTH 8086 Assembler..... 14
by Ray Duncan

Formal Definition of FORTH As a Programming Language 19
by C. H. Ting

Excuse Me, What Time Is It?..... 50
by Alan Bomberger

Background Information on Text Editors..... 55
by Leonard K. Nicholson

Standards for Computerized Scientific Measurement..... 57
by Douglas Crockford

Letters..... 6
Of Interest..... 8
Software Review.. 59
Book Reviews..... 60

Editor – Marlin Ouverson
Assistant Editor – Julie Anton
Contributing Editors – Dave Caulkins, Dave Cortesi, Kim Harris
Marketing Director – Craig S. Harper
Advertising Sales – Laird Foshay
Circulation – Ari Taylor, Terri Marty
Art Director – Clifford West
Production Manager – Barbara Ruzgerian
Typesetter – Renny Wiggins
Proofreader – Nancy Heubach
Cover Illustration – A. McCahon

© 1981 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, Box E, 1263 El Camino Real, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025.

EDITORIAL

Blue-sky thinking, the kind that opens the mind to new levels of possibility, brings with it many challenges. First, the imagination must break out of old patterns into unexplored territory. It must defeat the same critical self-consciousness which makes us feel embarrassed by our little non-conformities.

Next the mind itself is dared to entertain concepts which don't fit its preconceptions. This is where most blue-sky thinking gets doused with a strong dose of skepticism before it even has a shot at justifying itself. Knee-jerk censorship discards any notion not bearing a societal/peer/professional seal of approval – which, of course, any truly new idea cannot have obtained.

If some wonderfully chancy, ground-shaking, dimension-bursting thought makes it past all these rigid internal mechanisms of too-linear thinking, then it must pass a still greater challenge, that of finding even a small audience which will ponder it without haste or snickers. This is the last burial ground for blue-sky inspirations, here they die in the presence of those who have undertaken to preserve the technological state-as-it-is.

I believe that the unexplored territories of technology have not been diminished by the microcomputing revolution; rather, new frontiers are open to us. We are still sod-hut dwellers under the open sky, albeit connected by networks and terminals, by mutual interests in system tools, language development, and by the immensely powerful concept of personal computing. How will we nurture these things, and how will we use them to transform ourselves?

The only things hampering even greater progress than we have seen thus far, are found within our own creative processes and in the way we accept strange new worlds of thought when they impinge (too rarely) on our consciousness from without. *DDJ* will provide the forum for those who unleash their visions; ours is a company which began publishing personal computing periodicals before personal computers existed, before most people would seriously consider the idea. The stage is set . . . the sky's the limit! – *MDO*

Subscription Rates: $25 per year within the United States; $40 for first class to Canada and Mexico; $58 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Eric H. Johnson, Jerry Horn, Edward W. Carr, Steven Fisher, Thomas H. Grohne, G. V. Elkins, Robert M. Connors, William J. McEown, Robert C. Luckey, Kim Leary, Robert N. Stabler, DeWitt S. Brown, John B. Palmer, E. Glion Curtis, Jack Miller, Burks A. Smith, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Howard E. Decker, Jim Wilson, Frank Lawyer, Chris Pettus. **Retaining Subscribers:** Osamu Shimizu, Scott B. Guthery, Robert L. Edens, Titus Purdin, Steven Fisher Enterprises. **Sustaining Subscribers:** Gordon Eubanks, Jr. and Pete Roberts. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** LP Enterprises, 8/11 Cambridge House, Cambridge Road, Barking, Essex, 1G11 8NT, England (Tel. 01-591-6511, TLX 892395). Hofacker-Verlag, Tegernseer Strass 18, D-8150 Holzkirchen, West Germany. **Asia & Austraila:** Computer Store, POB 31-261, 22B Milford Road, Milford, Auckland 9, New Zealand. ASCII Publishing Co., Ltd., Aoyama Building 5F, 5-16-1 Minami Aoyama, Minato-ku, Tokyo 107, Japan.

LETTERS

Epson Salts

To the Good Doctor:

I would like to point out that cotrary to Alan Bomberger's article on "MX-80 to NorthStar Connections" (*DDJ* #35), I have found that directly making the connections to the NorthStar output port works perfectly.

The details:

1) The key point here is the duration of the output strobe pulse (active LO) that is made available by the NorthStar motherboard at the parallel port control header. A close look at the circuit diagrams shows that the pulse in question is between 1.5 and 2 "T" states long. At 4 MHz this works out to be between 0.375 and 0.5 micro-seconds. Well the ol' MX indeed requires 0.5 μs minimum thus we can't guarantee it! But guess what? It works flawlessly anyway.

2) Connect pin 4 to pin 12 on the port header; this directs our slightly fast strobe pulse to pin 8 of the output port connector on the back panel. This goes to pin 1 on the Epson.

3) Connect pin 3 to pin 13 on the header; this makes pin 7 on the output port connector the input line for the acknowledge from the MX (active LO). Connect pin 7 to pin 10 on the MX. This completes the handshaking signals.

4) Connect the 8 data lines and ground. I used ribbon cable without any twisted pairs of ground and data, and there is no data loss problem, though if the distance exceeded 10 feet or the environment was particulary noisy the twisted pairs might become necessary.

5) The ultra-simple input code as given in the NorthStar manual is all it takes to get that printer buzzing. A cash register printout tape will never sound the same!

With all respect to Alan, the output pulse is on the short side but for all those NorthStar users trying to make the connection to the Epson MX printer: the direct connection has not missed a bit in hundreds of listings.

A footnote to the NorthStar manual on the subject of port addresses. There are numerous references to IN 6 and OUT 6 as the appropriate instructions to access the motherboard status byte and control byte respectively. Please note that the port number instead of reading 6, should read 6 + port base. Long were the hours I couldn't get some simple code running when following their instructions because I had changed my port base address to accommodate some other board's hardwired port addressing and IN 6 or OUT 6 were connected to bizarre and unpredictable data.

Print On,
William Smith
P. O. Box 497
Valley Ford, CA 94972

MARCH Hare . . .

Dear Mr. Ouverson:

Here is clear proof of the soundness of your policy of requesting actual listings of programs. I had lots of excuses for not making one for my MARCH program (*DDJ* #62), but I'm sure we will all be sorry now. Line 12 of the program is missing. It should be RNZ ; return if the pattern is incorrect.

Otherwise the article looks as nice as the rest of the magazine, and I'm very proud to be a part of it. Thank you very much. Keep up the good work, and stick to that policy.

Regretfully,
James Laurino
414 South 47th Street
Philadelphia, PA 19143

Small-C Notes, Bug Fix

Dear Marlin,

Thank you for your kind words and your enthusiasm for my work with small-c. As time permits, I would like to continue writing for *DDJ*.

The answer to your question about the effect of my changes to the small-c expression analyzer is that some existing programs may need changing slightly. In such cases, however, the original code would be improperly written and would not work if compiled with another comiler. No doubt some users of small-c have discovered that when the compiler fails to double a value added to an integer array name, for instance, they can force the desired result by explicitly doubling the value in the program itself. Such code will not operate properly after the change.

I did encounter one situation, however, where a program adjustment avoided trouble with the old expression analyzer and also worked with the new one. The expression (*intptr - 1) would subtract 2 from the integer pointed to by intptr. The problem went away by writing (-1 + *intptr) instead. That is when I knew the binary plus and minus operators were not being handled symmetrically. Ironically, one analyzer bug (asymmetry) corrected the error of another (after applying the asterisk operator to intptr the result should no longer have appeared to be a pointer).

Since writing my article (*DDJ* # 62) about the expression analyzer, I discovered one more change that was needed. Line 387 of the listing should read:

```
387  lval[0] =lval[2] =0;
```

Sincerely,
J. E. Hendrix
Rt. 1, Box 74-B-1
Oxford, MS 38655

La Triviata (*sic*)

Dear People:

Once, long ago, I won some Latin contests in the state of Illinois. I never thought I'd have to use that knowledge again, but:

There is no such thing as HOMO-SAPIEN (Nicol, R. L., "Actors, Property Lists, and LISP," *DDJ* #62, p. 15). "Homo sapiens" is the scientific name of mankind. It means "wise man" (singular). "Wise men" would be "homines sapientes."

(A related "astounding fact" is that there is no such thing as a suburb, etymologically. The area near a city is its "suburbs" – singular. The areas (plural) near two cities (plural) are their "suburbes" (plural – three syllables, and the the last one pronounced "bees"). The city itself is the "urbs," and cities, plural, are "urbes.")

Sincerely,
W. D. Maurer, Professor
George Washington University
S. E. A. S.
Washington, DC 20052

Reclaim to Fame?

Dear Marlin,

I received your letter about the proposed new Small-C issue and am hoping to contribute something before your deadline. I have also contacted other people and some interest has been generated.

I have something that may be of use, and give your readers a useful program.

There have been several disk testing routines launched onto the market over the last few years. One of the first, and potentially most useful is RECLAIM

published by Lifeboat Associates. However, it was flawed by not working on all systems. In my case it was a Morrow Discus 2D. Through dissassembly I found the reason: the program makes direct calls on the BIOS when doing the actual testing of the disk surface. It makes the assumption that the zero flag is set on a return from read or write to indicate success. This is not so in all cases for the Morrow Cbios and others, I believe. The CP/M documentation confirms that no flag settings are used by Bios to indicate success – only that A is 0 for success, 1 otherwise.

I confirmed this by inserting a ORA A instruction immediately after the direct Bios read and write calls and having a successful run. It can also be fixed by the same patch in the Bios for those who do not wish to patch the program directly.

Another important point with disk testing routines is the built-in re-try count in the Bios. Re-tries may be great when trying to recover valuable data, but they can hide all the flaky sectors on the disk. I typically change this to 2 with the debugger, instead of the usual 10 before any testing. A good system seldom does a re-try anyway – I have used the system like this for hours with no bad sector errors.

I hope you find some use in this letter and hope to see more of your magazine soon.

Yours faithfully,
Angus S. Bliss
P. O. Box 293
Hamilton 2303
Australia

In Search Of . . .

Dear Sir:

I am looking for a "Tiny Basic" for use with the Signetics 2650 cpu. As a relative newcomer to small computers, I have not had any luck in finding the material I need to prepare a program for this obscure but quite workable cpu.

A study of one of your compendiums of early *Dr. Dobb's Journal*s indicates that a number of people were contributing articles on Tiny Basic for the 8080, and that you were calling for contributions for other microprocessors. Did anyone ever do anything with the 2650? If so, I would appreciate your help in being guided to the source of this material.

I have been examining Li-Chen Wang's Palo Alto Tiny Basic, which appeared in your May 1976 issue, with the idea of converting it over to 2650 language. This appears to be a formidable task, especially for a neophyte. Because of this, I would be happy to hear from you any comments on this matter. I would also be happy to correspond with any of your readers who are knowledgable on the 2650.

Thank you for your attention.

Sincerely,
George A. Luce
2907 Greenlawn Pkwy.
Austin, TX 78757

DDJ

OF INTEREST

by D. E. Cortesi

April Showers . . .

Another batch of show announcements turned up this month, enough to fill all the empty spaces in your spring calendar. The second annual Eighty/Apple show, a show for TRS-80 and Apple users, will take place in New York on April 2-4. Contact **Kengore** for details.

The **National Computer Shows** organization will stage expositions of computers and office equipment at three sites this spring: Dallas, TX April 15-18, the Nassau Coliseum on Long Island April 22-25, and in Los Angeles, CA, May 6-9.

John Craig's **Computer Swap America**, after a February 6 swap meet in Costa Mesa, CA that will probably be over when you see this, will return to San Jose, CA on April 24th. Craig is prepared to offer the use of his trademark and a complete promotion package to "qualified individuals or organizations" that want to put on their own swap meets.

. . . and a November Show

The California Educational Computing Consortium (whew!) will hold their sixth annual conference in San Diego, CA on November 18-19. There is still time to submit a paper dealing with computers and applications of interest to college-level educators; send a draft by March 1 to **Professor Frances Grant**.

Radio Shack Info

Registered TRS-80 users receive Tandy's newsletter, *TRS-80 News*. The first 20 issues of that paper, through the December 1980 number, have been reprinted in a $5 paperback that may be found at Radio Shack dealers.

There are other sources of TRS-80 information. **Kengore** has started the *Eighty System Newsletter*, a weekly for "users, retailers, distributors, or manufacturers of TRS-80 related products." Each issue is said to contain both late news of "the TRS-80 industry" and a bibliography of related articles from "over 100" other publications. An annual subscription is $39.

Soft Wares

The **Software Toolworks** has updated its C/80 compiler (originally based, we believe, on Cain's Small-C). The latest version generates assembly source that is compatible with both the standard ASM assembler and with Microsoft's Macro-80 relocatable assembler. An execution profile facility – a means of mapping the frequency of use of each program statement, and a priceless development tool – has been added to the runtime library. The package is an outstanding buy at $40 plus shipping; present owners can have an update for $10.

Scientists, engineers, and educators who use Apple Pascal should look into the DIFF E-Q package from **Sage Software**. It's basically a set of programs for solving differential equations up to the 24th order, but it also contains high-resolution graphics operations for displaying families of curves. That, the makers claim, makes it "ideal for lecture demonstrations."

High-resolution plots in another medium can be had with AccuChart from **Datagraph**. This CP/M program writes graphs, bar charts, and pie charts using a Diablo (-compatible) daisy printer. The data to be plotted may be entered from the keyboard or taken from a disk file. The price is $295.

Miken Optical have implemented an unusual idea for CP/M – an interactive BIOS. IBIOS apparently adds a rudimentary command interpreter to the BIOS, upstream of the rest of the system. With it in place, you can perform some useful functions at any time, regardless of what's going in the transient program area. The $60 program is distributed as 8080 source, and must be customized to your system.

Harder Wares

The AIM, that endlessly-useful little machine, is the base for two new hardware products. One, from **Columbus Instruments**, is a board that carries sixteen channels of 12-bit analog-to-digital conversion, a clock-calendar, and four "universal" memory sockets. An AIM so equipped, they say, is a complete system for laboratory data acquisition and logging. The A/D conversion channels operate in 25 microseconds; the clock has battery backup; and when equipped with 2764 or equivalent chips, the board provides 32K bytes. The full board costs $1270 (ouch) but the A/D and clock can be left off to reduce the price.

The second AIM add-on is an SS-50 bus from **Percom**. The AIM's expansion pins plug into an interface board from which extends an SS-50 bus motherboard. With it, according to the release, users "can add disk storage, a CRT, and other . . . System-50 modules available from several vendors." The board costs $90.

Comdex Extravaganza

To judge from the press release pile, everyone and his dog announced something at the Comdex show. Here is a sampler of the more interesting ones.

Tecmar was the first to come out with add-ons for the IBM Personal Computer: hard disk, clock-calendar, voice output, BSR X10 driver, IEEE 488 driver, A/D and D/A converters, etc. Oh, and an expansion chassis to plug them all into (*An expansion chassis?* This is where we came in!).

Alspa Computer used the new, skinny, 8-inch drives to squeeze a two-drive, 64K, Z80 system into a unit rather smaller than a breadbox (about 6 x 9 x 17 inches), with an attractive $3000 pricetag.

Dynabyte's newest unit is a little larger, but it claims to support up to eight concurrent MP/M or Oasis users at just over $1000 per user. Their model 5505 contains a 16MB, 5-inch hard disk, a 1MB 5-inch diskette drive for backup, and "400K bytes" of RAM.

Did Dynabyte's 5505 frighten **Altos**? For whatever reason, the latter company announced price reductions of about $1000 on their 8000-2 and 8000-15 models.

Well, both those companies had better check out **CompuThink**'s Hawk-32 model, a multi-user system with a 5-inch hard disk and a 68000 CPU, to be sold for "under $10,000." This system runs yet another UNIX look-alike operating system.

Net Works

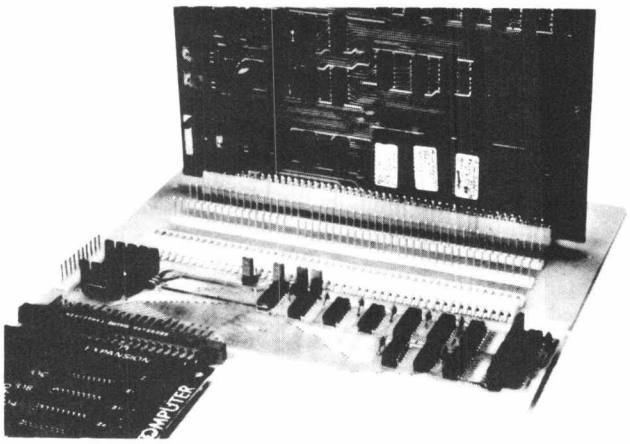
A press release from **Stanford University** passes on Nobel laureate Joshua Lederberg's praise for computerized text systems. Stanford is extensively wired, with terminals in most buildings and several related time-sharing services in operation. Lederberg notes the usual advantage of word processing ("You are really in close touch with your literary product . . . . You can go through an indefinite number of drafts . . .") but finds equal or greater value in the bulletin-board style of communication. "We found that internal communication on the Stanford campus, separated by not more than a few hundred yards, was in fact greatly facilitated by handling a great deal of our discourse over the wire. It

ended up we did 95 percent of our communications, . . . the elaboration of ideas, getting criticisms about proposals, over the wire rather than the telephone. Rarely did we ever do a written communication."

"Terminals of this kind and their searching capabilities are going to give one access to the libraries of the world," Lederberg said, but he wants more: "You should be able to write marginal notes on the stuff you have read, looked at, and wish to criticize. Since you can do this nondestructively, you can be helpful in informing other users of the same library in a way that would not be tolerated at the present time."

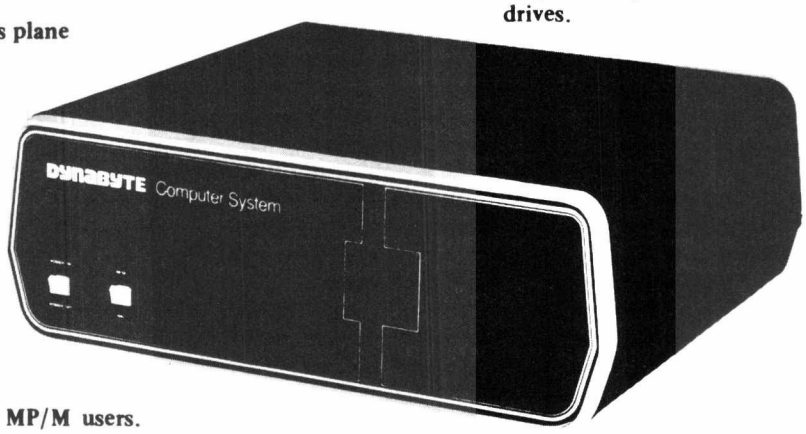
We don't know if Dr. Lederberg got that last idea from Ted Nelson's **Project Xanadu**, but that is just the kind of thing that Nelson describes in his book *Literary Machines*, available from the Project for $15. The Xanadu (tm) system, Nelson claims, exists in prototype form. It allows any user to make nondestructive notes in the margin of a retrieved document, and has a number of other remarkable properties as well. We have a copy, and it's a fascinating read, guaranteed to fill any computerist's head with dazzling visions and daring possibilities.

DDJ



Percom's board grafts an SS-50 bus plane onto an AIM.

Alspa Computer uses slim-line 8-inch drives.



The Dynabyte 5505 supports eight MP/M users.

Contact Points

Alspa Computer, Inc., 300 West Harvey Blvd., Santa Cruz, CA 95060; (408) 429-6000.

Altos Computer Systems, 2360 Bering Drive, San Jose, CA 95131; (408) 946-6700.

Columbus Instruments International Corporation, 950 N. Hague Ave., Columbus, OH 43204; (614) 488-6176.

Computer Swap America, POB 52, Palo Alto, CA 94302; (415) 494-6862.

CompuThink, 965 West Maude Ave., Sunnyvale, CA 94086; (408) 245-4033.

Datagraph, P. O. Box D, Arnold, CA 95223; (209) 795-5104.

Dynabyte, 521 Cottonwood Drive, Milpitas, CA 95035; (408) 263-1221.

Professor Frances Grant, Center for Information and Communications Study, California State University, Chico, CA 95929.

Kengore Corporation, 3001 Route 27, Franklin Park, NJ 08823; (201) 297-2526.

Miken Optical Company, 53 Abbett Avenue, Morristown, NJ 07960; (201) 267-1210.

National Computer Shows, 824 Boylston Street, Chestnut Hill, MA 02167; (617) 739-2000.

Percom, 11220 Pagemill Road, Dallas, TX 75243; 800-527-1222.

Project Xanadu, Box 128, Swarthmore, PA 19081.

Sage Software Company, 1322 La Loma Ave., Berkeley, CA 94708; (415) 843-0863.

Software Toolworks, 14478 Gorietta Drive, Sherman Oaks, CA 91423; (213) 986-4885.

Stanford University News Service, Stanford, CA 94305; (415) 497-2558.

Tecmar, Inc., 23600 Mercantile Road, Cleveland, OH 44122; (216) 464-7410.

Recursion in FORTH

Recursion, which is the ability of a procedure to use itself (either directly or indirectly) as a subprocedure, is regarded by most programmers as a fundamental requirement of programming languages. Not only are many mathematical functions most conveniently expressed in recursive form; but there is a large number of non-numeric operations that can most conveniently be expressed recursively. This is particularly true of operations on list or tree structures. Thus, any operation on a tree can be expressed by the following algorithm:

1. Perform the operation on the root of the tree
2. Perform the operation in turn on all the subtrees.

A recursive definition consists of two parts:

1. A definition of how the procedure is to be applied to a terminal value.
2. A definition of how the procedure is to be applied to an arbitrary value, expressed in terms of the same procedure applied to the predecessor of that value. The predecessor of a tree is regarded as the set of subtrees that are left when the root is removed. The predecessor of a positive argument of a numeric function is that argument diminished by unity. Thus we may define a recursive schema for a function F operating on positive integers as follows:

```
F(n) = A                (n = 1)
     = G(n, F(n'))      (n > 1)
```

where A is a constant, G is a second function that may be defined independently of F, and n' is the predecessor of n. To take a simple example, the factorial function may be defined

```
F(2)      = 2
F(n > 1)  = n * F(n-1)
```

in which the multiply operator plays the part here of the function G in the schema above.

by Glyn Emery

Glyn Emery, Dept. of Computer Science, Adran Gwyddor Cyfrifiadur, University of Wales, Aberystwyth, Coleg Prifysgol Cymru, Wales SY23 3BZ

The above procedures are auto-recursive, or self-recursive; i.e., they are defined directly in terms of themselves. There is, however, a class of procedures that are defined in terms of a second procedure, which in turn is defined, either directly or indirectly, in terms of the first. This form of definition is known as mutual recursion. In many cases mutual-recursion definitions can be reduced to cases of self-recursion; but there are functions that can be defined by mutual recursion but not by self-recursion. Nevertheless, even when a reduction to self-recursion can be carried out, it may not be convenient, since doing so may obscure the algorithm and make a program more difficult to develop.

Although FORTH functions are evaluated using a stack mechanism, which is a fundamental requirement for recursive programming, FORTH unfortunately cannot provide users directly with the ability to use it recursively. The possibility of self-recursion is explicitly excluded from FORTH by the smudging process, which ensures that a word cannot be found in the dictionary until its definition has been completed. Smudging inverts the bit in the "32" position of the length byte in the name field at the start of compilation. In the case of function definitions, this bit is returned to its original value only when the semicolon operator is obeyed at the end of the compilation. Throughout the compilation, therefore, the length byte is "wrong" and the word cannot be matched unless, of course, we are performing a redefinition of a word that is already present in the dictionary.

Nor can mutual recursion be achieved in FORTH. This is because any reference within a FORTH definition must be to a word that has already been defined and is already present in the dictionary. This restriction does not apply to other high-level langauges, since most compilers operate in two or more passes. An early pass is used to assign locations to identifiers: a subsequent pass enters references to these locations wherever they are appropriate. We might note here that, were it possible to achieve mutual recursion in FORTH, then it would be possible to achieve self-recursion in spite of the smudging process, since function F could refer internally to function G, which in turn could be defined as identical to F.

One technique for achieving self-recursion is to plant the code-field address of the word in the appropriate position in its own parameter field by some means other than the normal dictionary search. Fortunately, this can be done without much difficulty, since a new word is added to the current vocabulary right at the start of its compilation. The definition in Figure A defines a word that will place on the stack the code-field address of the last word added to the vocabulary. This word must be declared immediate, since we shall require it to do its work during compilation. Notice that two "contents" operators are required, since CURRENT actually points to a word within the vocabulary definition itself, and this in turn points to the last word defined.

The word LITERAL, which is also immediate, if obeyed during compilation, places the top stack item in the next available space in the dictionary, and will therefore compile the address generated by THISCODE. At execution time, this address will simply be transferred to the stack, and can therefore be executed by obeying the word EXECUTE. Thus, a complete recursive definition of the factorial function would be as found in Figure B.

An alternative way of performing self-recursion is to define the desired function initially as a dummy. See Figure C. The function can now be used in the definition of itself, except that, to avoid redefinition, we must give the full definition an alias. An example is found in Figure D. In the dummy definition, 0 ensures that a single 16-bit word is allotted, provided of course that 0 is a real dictionary entry, as it is in most dialects of FORTH. The final requirement is to construct a thread to lead from FAC to FACALIAS. In a sense this is a renaming, so we might write

```
RENAME  FACALIAS  FAC
```

where RENAME is defined as in Figure E. The "tick" operator is immediate, so [COMPILE] must be used to make sure that it is compiled into the definition of RENAME. Notice that it is [COMPILE] and not COMPILE that must be used here. This is because the tick must be forced to execute in compilation mode, not in execution mode, since in the latter mode it operates in the same way as the word LITERAL and so in this instance would merely stack the code-field addresses of CFA and !. Treated in this way, the "tick" operators ensure that the

Figure A

```
: THISCODE CURRENT @ @ PFA CFA ; IMMEDIATE
```

Figure B

```
: FAC DUP 2 = IF ELSE DUP 1 - THISCODE LITERAL EXECUTE
                      * ENDIF ;
```

Figure C

```
: FAC 0 ;
```

Figure D

```
: FACALIAS DUP 2 = IF ELSE DUP 1 - FAC * ENDIF ;
```

Figure E

```
: RENAME [COMPILE] ' CFA [COMPILE] ' ! ;
```

Figure F

$$f(x+1) = x + g(x) + f(x) \qquad (x > 1)$$
$$g(x+1) = 2f(x) + 1 \qquad (x > 1)$$
$$f(1) = 1$$
$$g(1) = 0$$

Figure G

```
: EFF 0 ;
: GEE 1 - DUP 0 > IF EFF 2 * 1+ ENDIF ;
: FALIAS 1 - -DUP 0= IF 1 ELSE DUP GEE OVER EFF
                        + + ENDIF ;
RENAME FALIAS EFF
```

Figure H

```
: DUPS DUP 0 DO 0 SWAP LOOP SP@ 2+ SWAP DUP + OVER OVER
     + ROT ROT CMOVE ;
```

Figure I

```
: DROPS 0 DO DROP LOOP ;
```

Figure J

$$n(x, 2) = x + 1$$
$$n(x, y+1) = q(x,y,x) \qquad (y > 1)$$
$$q(x,y,0) = 1$$
$$q(x,y,s) = 2\ v(x,y,s-1)\ q(x,y,s-1) \qquad (s \geq 1)$$
$$v(x,y,0) = n(x,y)$$
$$v(x,y,s) = n(x^{q(x,y,s)}, y)$$

Figure K

```
: EXP OVER SWAP -DUP 0= IF ELSE 0 DO OVER * LOOP
     ENDIF SWAP / ;
: QQQ 0 ;
: NNN DUP 2 = IF DROP 1+ ELSE 1 - OVER QQQ ENDIF ;
: NU -DUP 0= IF NNN ELSE 3 DUPS QQQ >R DROP SWAP
   R> EXP SWAP NNN ENDIF ;
: QALIAS -DUP 0= IF 2 DROPS 1 ELSE 1 - 3 DUPS
   NU >R QQQ R> * DUP + ENDIF ;
RENAME QALIAS QQQ
```

Figure L

```
: TRACE EMIT SPACE DUP 1+ DUPS 0 DO . LOOP DROP ;
```

code-field address of FACALIAS is stored in the parameter field of FAC in place of the 0.

The same technique can be used to achieve mutual recursion. Consider the recursive system found in Figure F. The function f could in fact be expressed in a self-recursive schema; but this example will do for present purposes. In FORTH, the system becomes as in Figure G.

The examples considered so far have involved functions with only one argument, so that a simple DUP has been sufficient to provide a copy for a recursive call. For general recursive use it is convenient to provide a word that will "pass" an arbitrary number of parameters to any called function by making copies of them on the stack. The word found in Figure H does this; so that the call

```
n DUPS
```

makes a copy of the top n stack items (excluding n itself). Another useful word in this connection is shown in Figure I.

In Khinchin[1] a recursive function n(x,y) is defined in a manner equivalent to that found in Figure J.

Thus n(x,y) is defined for all $x \geq 1$ and all $y \geq 2$. This function cannot in fact be reduced to a case of self-recursion, though it can be expressed more simply than this; but, using the techniques developed above, we can easily evaluate it in FORTH. See Figure K.

Unfortunately only a few values of this function can be evaluated in single-length arithmetic. The same words could, however, be used in double or multiple length, if the arithmetic and stack operators were suitably redefined.

A useful operator to use when developing FORTH functions is found in Figure L. This outputs the top stack item as an ASCII character, followed by a number of stacked parameters. Thus,

```
81 3 TRACE
```

outputs a Q followed by the top three stack items (not counting the 81 and the 3).

DDJ

Reference

1. Khinchin, A. Y. *Three Pearls of Number Theory*, Gaylock Press, 1952, p. 13.

FORTH 8086 Assembler

Focus on FORTH

IBM's selection of the Intel 8088 as the heart of their Personal Computer has ensured that the 8086/8088 family will be a hot item for years to come. Many companies (including my own) are preparing FORTH program development systems and application programs to run on these machines. Although FORTH assemblers are a vital part of the FORTH programming environment, writing and debugging them is a relatively bleak task. We are placing the accompanying code in the public domain so that FORTH programmers can use their time more profitably elsewhere.

FORTH assemblers are used in two distinct ways. The first is as an integral part of the FORTH runtime system for creation of new machine language primitives in the FORTH dictionary. This capability is useful in applications such as process control that are highly time dependent; many FORTH programs can be speeded up to a surprising degree by recoding a few frequently used words (such as the components of inner loops) in assembly language. The other common application for an assembler is metacompiling; that is, the generation of FORTH programs on a host processor for a different target processor.

Paradoxically, although FORTH assemblers are crucial to both the power and transportability of the FORTH language, they tend to be poor examples of "good" FORTH programming style. The necessary decoding of address modes and manipulation of bit fields results in code that is complex, terse, and obscure – in contrast to FORTH application programs, where the highest level words have simple, short definitions that describe exactly what they do.

This 8086 assembler was originally prepared to be embedded in the Nautilus Systems Cross Compiler, and was used to transport FORTH onto the Lomas Data Products 8088 microcomputer. The version presented here is written in standard fig-FORTH for use as part of a FORTH runtime package. Due to the many addressing modes and the highly asymmetric architecture of the 8086/8088 family, the code is bulky and includes a generous amount of error-checking logic compared to traditional FORTH assemblers. For those readers who wish to cut down the memory space requirements, a few suggestions are included at the end of the article.

In spite of the assembler's syntax checking and error reporting, it is still easy to produce garbage code and crash the FORTH interpreter/compiler. For this reason, when one is writing assembler code, it should be edited into disk screens and loaded in small modules for debugging. Be careful to preserve the interpretive pointer and return stack pointer across FORTH words (their actual location is implementation dependent, see the installation manual for your FORTH system). Modification of these registers will result in an immediate crash of the interpreter/compiler.

by Ray Duncan

Ray Duncan, c/o Laboratory Microsystems, 4147 Beethoven Street, Los Angeles, CA 90066.

Using the Assembler

Each new word definition must begin with the word CODE, followed by the new FORTH name. This sequence creates a dictionary header with the contents of the code field address pointing to the parameter field so that when the word is invoked, control is transferred by the inner interpreter directly to the machine language routine. The ASSEMBLER vocabulary is automatically selected. A series of mnemonics and operands may then be entered in postfix notation. The entire definition must be closed by the word NEXT, which does some error checking, reselects the FORTH vocabulary, and encodes an unconditional branch back to the inner interpreter.

Here is a simple example using the Assembler to create a fast integer multiply by ten routine (expects the argument on top of stack, and leaves the result on top of stack; see Figure 1).

Execution control when coding assembler primitives is provided through the use of BEGIN . . . UNTIL, IF . . . ENDIF, and IF . . . ELSE . . . ENDIF constructs very similar to those in the usual FORTH vocabulary. The primary difference is that the assembler versions of these constructs operate on the basis of the CPU status flags rather than the top of stack parameter; they do not change the CPU flags or the contents of the parameter stack. Here is an example of a loop which repeatedly reads port #1, waiting for bit #2 to be set (See Figure 2).

A more general purpose version could be coded which would accept a port number and bit mask on the stack (See Figure 3).

The assembler follows the Intel mnemonic as closely as possible, with the following exceptions:

- Postfix notation. Operands *precede* the instruction mnemonic. Destination operands are terminated by a comma. For example:

```
AX, DX  MOV
```

  means copy the contents of register DX into register AX.

- Parts of the operand must be separated by spaces. For example:

```
AX, DX  MOV
```

  is legal, while

```
AX,DX   MOV
```

  will cause an assembler error.

- Immediate data values are designated with the "#" operator, similar to the DEC macro-assembler. For example:

```
AX, # 0001      MOV
```

  means load the value 0001 into the AX register, while

```
AX, 0001        MOV
```

  indicates load the contents of location 0001 into the AX register.

- When a numeric value occupies the destination operand field, it must be followed by a free-standing comma. For example, to store the contents of the AX register into memory location 010A, you would write:

```
010A , AX       MOV
```

- When indexed addressing is used, any displacement must precede the index register specification. For example:

```
AX, 4 [BX]      MOV
```

  means load the contents of the address pointed to by register BX plus 4 into the AX register.

- In general, the assembler relies on flags set by the operands to determine the addressing mode. In a few special cases the W (word/byte) flag cannot be inferred from the operand field and must be explicitly set with the BYTE or WORD operators. If not set by the

programmer, the mode defaults to byte. Examples:

```
BYTE     LODS

WORD     STOS

BYTE 5 [BX]   NEG
```

- The following instructions are not implemented: ESC, INT, LDS, LES, and SEG.
- Memory direct addressing with the arithmetic or logical instructions such as ADD and OR is not supported at present. However, all varieties of direct addressing involving the MOV instruction are supported.
- When coding jumps, conditional branches, or calls, you must supply the absolute offset address. The assembler calculates the displacement and issues an error message if the target address is out of range.
- Intersegment "long" jumps and calls are not supported.
- In order to keep the assembler from becoming even more bulky, certain special case machine code sequences are not supported. This causes the assembled code to be slightly longer in a few instances but does not result in any logical difference at execution time. For example, the 8086 instruction set includes a special CMP instruction for comparison of AX with immediate data, and a more general purpose but slightly longer CMP instruction to compare any register with an immediate data value; only the latter is implemented in this assembler.

```
CODE      PORT-WAIT
          BEGIN
AL, 01    IN              ( read port #1 )
AL, # 02  TEST            ( check if bit 2 set )
          UNTIL           ( loop until Z flag false )
          NEXT
```

Figure 2

```
CODE      PORT-WAIT
DX        POP             ( port number )
BX        POP             ( bit mask )
          BEGIN
AX, DX    IN              ( read port )
AX, BX    TEST            ( check status bit )
          UNTIL           ( loop until Z flag false )
          NEXT
```

Figure 3

Operation of the Assembler

Because of the generality of the 8086 instruction mnemonics, the addressing mode usually cannot be inferred from the instruction itself, as it can in the 8080 (e.g., LXI, LHLD). Consequently, the assembler relies on the operands for this information. Each register-related source or destination operand is defined through a BUILDS . . . DOES statement, so that at execution time they will appropriately set up the variables containing the word/byte flag, the direction flag, any indexing offsets, addressing type, and the register codes. In those cases where an operand field is not a register or register index, other operators ("#" and ",") are provided to set the flags.

Because of the FORTH postfix notation, when the instruction mnemonic is executed all information is already available in the variables for it to assemble the final machine code. The final assembly itself can be very complex. For example, the MOV instruction must be able to hand over 25 legal addressing combinations while screening out impossible transfers (such as trying to copy an 8-bit register into a 16-bit register).

The 8086 instructions are divided into fourteen groups for assembly processing based on the structure of the machine code:

1. Simple single byte instructions (no operand).
2. Conditional branches and loops with 1 byte offset.
3. String instructions CMPS, LODS, MOVS, SCAS, and STOS.
4. PUSH and POP.
5. Intrasegment jumps and calls of the form "JMP disp16" or "JMP reg/mem".
6. IN and OUT.
7. Two operand arithmetic/logical instructions: ADC, ADD, AND, CMP, OR, SBB, SUB, TEST, and XOR.
8. Single operand arithmetic/logical instructions: DIV, IDIV, IMUL, MUL, NEG, and NOT.
9. Various shifts: RCL, RCR, ROL, ROR, SAL, SAR, SHL, and SHR.
10. Simple two byte instructions (no operand).

```
CODE      FAST10*
AX        POP
BX, AX    MOV
AX, AX    ADD             ( * 2 )
AX, AX    ADD             ( * 4 )
AX, BX    ADD             ( * 5 )
AX, AX    ADD             ( * 10 )
AX        PUSH
          NEXT
```

Figure 1

11. INC and DEC.
12. Various MOV instructions.
13. Various XCHG instructions.
14. TEST instruction.

A number of internal primitives are used by the assembler to produce machine code. Ordinarily, these are not used by the FORTH application programmer, but their operation is described in Table I to facilitate extension and modification of the assembler itself.

Pruning the Assembler

Those readers who need to reduce the size of the assembler module may consider the following suggestions:

1. Remove error checking code.

2. Replace the CASE statements with Kim Harris' Case-Colon constructs or nested "IF . . . ELSE . . . THEN" constructs. Alternatively, use a machine language implementation of "OF" (see Dr. Eaker's original article for an example).

3. Remove infrequently used instructions and addressing modes altogether.

4. Modify the register BUILDS . . . DOES definition so that it increments a counter; from this it can detect whether it is processing the source or destination register. This would allow you to eliminate one set of register names and one of the register processing routines. ■■J

(Listing begins on page 33)

Table I

!<SP>	Set the local stack pointer to the current parameter stack position.
?<SP>	Leave the depth of the local stack on top of the stack. Used by the register decoding routines to determine whether a displacement has been specified.
?W	Leave the "W" (word/byte) flag on top of stack.
?TD	Leave the internal addressing indicator of the destination operand on top of stack. This takes on a value in the range 0-5 which may be interpreted as follows: 0 Direct addressing 1 Immediate operand 2 8-bit register 3 16-bit register 4 Base relative indexed addressing 5 16-bit segment register
?TS	Leave the internal addressing indicator of the source operand on top of stack. This takes on a value of 0-5 as above.
?RD	Leave the code 0-7 of the destination register on top of stack.
?RS	Leave the code 0-7 of the source register on top of stack.
?OD	Leave the displacement of the destination operand on top of stack. Only valid for the direct indexed and base relative indexed addressing modes.
?OS	Leave the displacement of the source operand on top of stack. Only valid for the direct indexed and base relative addressing modes.
RESET	Clear all assembler flags and modes.
OFFSET8,	Calculate the displacement from the address on top of stack to HERE and store it as an 8-bit field into the assembled code. Used by conditional branches and loops. If the branch is out of range, issue an error message.
OFFSET16,	Calculate the displacement from the address on top of stack to HERE and store it as a 16-bit field into the assembled code.
DISP,	According to the operand type and displacement value, set the MOD field, store the forming byte on top of stack into the assembled code, then store an eight- or sixteen-bit displacement value if necessary.
DSET	Check whether the source or destination operand uses base relative indexed addressing and set the direction flag accordingly.
BYTE	Force the W flag to indicate byte mode.
WORD	Force the W flag to indicate word mode.

The following primitives assemble flag fields into the lower byte of the word on top of stack:

+W	Merge the word/byte flag with the forming byte.
+D	Merge the direction flag with the forming byte.
+RD	Merge the destination register code with the forming byte.
+RS	Merge the source register code with the forming byte.
MOD1	Set the mod field to the bit pattern 01.
MOD2	Set the mod field to the bit pattern 10.
MOD3	Set the mod field to the bit pattern 11.

Formal Definition of FORTH

Focus on FORTH

FORTH was developed as a programming tool to solve real-time control problems.[1] While it has never been formally defined as a programming language, I think FORTH is mature enough now that it can be defined rigorously. The wide-spread use of this powerful tool requires that a common base be established to facilitate the exchange of programs and ideas in a standardized language form. The publication of FORTH-79 Standard[2] clearly reflects this necessity. To define FORTH as a programming language also helps us to focus our attention on the basic characteristics of FORTH, and to understand it more fully.

This article will present the definition of FORTH in the Backus Normal Form (BNF) notation. The basic syntax is presented in Table I, in which the focal point is the definition of "word". Some detailed clarifications on colon definitions and defining words are worked out in Tables II and III. Explanatory notes are arranged by section to highlight some problems not clearly expressed in the formal definitions.

Programming Language

A programming language is a set of symbols with rules (syntax) of combining them to specify execution procedures to a computer. A programming language is used primarily to instruct a computer to perform specific functions. However, it can also be used by programmers to document and to communicate problem solving procedures. The most essential ingredients of a programming language are therefore the symbols it employs for expressions and the syntax rules of combining the symbols for man-machine or man-man communications.

FORTH uses the full set of ASCII characters as symbols. Most programming languages use subsets of ASCII characters, including only numerals, upper-case alphabets, and some punctuation characters. Use of punctuation characters differs significantly from language to language. Non-printable characters are generally reserved exclusively for the system and are not available for language usage. In employing the full ASCII set of characters, FORTH thus allows the programmer a much wider range of usable symbols to name objects. On the other hand, the prolific use of punctuation characters in FORTH makes comprehension very difficult by uninitiated programmers.

Only four of the ASCII characters are used by FORTH for special system functions and are not for programming usage: NUL (ASCII 0), RUB (ASCII 127), CR (ASCII 13), and SP (ASCII 32). RUB is used to delete the previously entered character. It is used at the keyboard interactively to correct typing errors. NUL, CR, and SP are delimiting characters to separate groups of characters to form words. All other characters are used to form words and are used the same way. Non-printable characters are treated the same as printable characters. Because non-printable characters are difficult to document and communicate, their usage is discouraged in normal programming practice. However, the non-printable characters are very useful in maintaining a secured system. For terminals that have function keys, their control sequences may be used as the names of the operations each performs.

Words

Words are the basic syntactical units in FORTH. A word is a group of characters separated from other words by delimiting characters. With the exception of NUL, CR, SP and RUB, any ASCII character may be part of a word. Certain words for string processing may specify a regular character as the delimiting character for the word immediately following it, in order to override the delimiting effect of SP. However, the delimiting effect of CR and NUL cannot be overridden.

The usage of "word" in FORTH literature is very confusing because many quite different concepts are associated with it. Without sorting out these different aspects of "word" into independently identifiable entities, it is impossible to arrive at a satisfactory description of this language. Here the word is defined as a syntactical unit in the language, simply a group of characters separated from other words by delimiting characters. Semantically (concerning the meaning of a word), a word in FORTH can be only one of three things: a string, an instruction, or a number.

A FORTH program is thus simply a list of words. When this list of words is given to a computer with a FORTH operating system loaded in, the computer will be able to execute or interpret this list of words and perform functions as specified by this list. The functions may include compilation of new instructions into the system to perform complicated functions not implemented in the original operating system.

A string is merely a group of characters to be processed by the FORTH computer. To be processed correctly, a string must be preceded by an instruction which specifies exactly how this string is to be processed. The string instruction may even specify a regular character as the delimiting character for the following string to override the effect of SP. It is often appropriate to consider the string to be an integral part of the preceding instruction. This would disturb the uniform and simple syntax rule in FORTH and it is better to consider strings as independent objects in the language.

String processing is a major component in the FORTH operating system because FORTH is an interpretive language. Strings are needed to supply names for new instructions, to insert comments into source text for documentation, and to produce messages at run-time to facilitate human interface. The resident FORTH instructions for string processing are all available to programmers for string manipulations.

A number is a string which causes the FORTH computer to push a piece of data onto the data stack. Characters used in a number must belong to a subset of ASCII characters. The total number of characters in this subset is equal to a "base" value specified by the programmer. This subset starts from 0 and goes up to 9. If the base value is larger than 10, the upper-case alphabets are used in their natural sequence. Any reasonable base value can be specified and modified at run-time by the programmer. However, a very large base value causes excessive overlapping between numbers and instructions, and a "reasonable base value" must avoid this conflict in semantical interpretation.

A number can have a leading "-" sign to designate data of negative value. Certain punctuation characters such as "." are also allowed in numbers depending upon the particular FORTH operating system.

The internal representation of numbers inside the FORTH computer depends upon implementation. The most common format is a 16-bit integer num-

by C.H. Ting

C. H. Ting, Offete Enterprises, Inc., 1306 S. B St., San Mateo, CA 94402.

ber. Numbers are put on the data stack to be processed. The interpretation of a number depends entirely on the instruction which uses the number. A number may be used to represent a true-or-false flag, a 7-bit ASCII character, an 8-bit byte, a 16-bit signed or unsigned integer, a 16-bit address, etc. Two consecutive numbers may be used as a 32-bit signed or unsigned double integer, or a floating point number.

FORTH is not a typed language in which numerical data types must be declared and checked during compilation. Numbers are loaded on the data stack (hereafter called the "stack") where all numbers are represented and treated identically. Instructions using the numbers on the stack will take whatever they need for processing and push their results back on the stack. It is the responsibility of the programmer to put the correct data on the stack and use the correct instructions to retrieve them. Non-discriminating use of numbers on the stack might seem to be a major source of errors in using FORTH for programming. In practice, the use of the stack greatly eases the debugging process in which individual instructions can be thoroughly exercised to spot any discrepancies in stack manipulations. The most important advantage gained in the uniform usage of data stored on the stack is that the instructions built this way can be context-free, allowing them to be repeatedly called in different environments to perform the same task.

Numbers and strings are objects or nouns in a programming language. Typed and named numbers in a program provide vital clues to the functions and the structures in a program. The explicitly defined objects or nouns make statements in a program easy to comprehend. The implicit use of data objects stored on the data stack makes FORTH programs very tight and efficient. However, statements in a program deprived of nouns are difficult to understand. For this reason, the most important task in documenting a FORTH program is to specify the stack effects of the instructions, indicating what types of data are retrieved from the stack and what types of data are left on the stack upon exit.

Standard Instructions

In a FORTH computer, an instruction is best defined as "a named, linked, memory resident, and executable entity which can be called and executed interactively." The entire linked list of instructions in the computer memory is called a "dictionary." Instructions are known to the programmer by their ASCII names. The names of the instructions in a FORTH computer are words that a programmer can use either to execute the instruction interactively or to build (compile) new instructions to solve his programming problems.

In FORTH literature, instructions are called "words," "definitions," or "word definitions." The reason that I chose to call them "instructions" is to emphasize the fact that an instruction given to the FORTH computer causes immediate actions performed by the computer. The instructions in the dictionary are an instruction set of the FORTH virtual computer, in the same sense as the instruction set of a real CPU. The difference is that the FORTH instructions can be executed directly and the FORTH instructions are accessed by their ASCII names. Therefore, FORTH can be considered as a high-level assembly language with an open instruction set for interactive programming and testing. The name "instruction" conveys more precisely the characteristics of a FORTH instruction than "word" or "definition," and leaves "word" to mean exclusively a syntactical unit in the language definition.

Instruction set is the heart of a computer as well as of a language. In all conventional programming languages, the instruction set is immutable and limited in number and in scope. Programmers can circumvent the shortcomings of a language by writing programs to perform tasks that the native instruction set is not capable of. The instruction set in a FORTH computer provides a basis or a skeleton from which a more sophisticated instruction set can be built and optimized to solve a particular problem.

Because the instruction set in FORTH can be easily extended by the user, it is rather difficult to define precisely the minimum instruction set a FORTH computer ought to have. The general requirement is that the minimum set should provide an environment in which typical programming problems can be solved conveniently. FORTH-79 Standard suggested a convenient instruction set as summarized in Table I. These instructions provided by the operating system are called "standard instructions," and are divided into nucleus instructions, interpreter instructions, compiler instructions, and device instructions.

User Instructions

Instructions created by a user are called "user instructions." There are several classes of user instructions depending upon how they are created. High level instructions are called "colon instructions" because they are generated by the special instruction ":". Low level instructions containing machine codes of the host CPU are called "code instructions" because they are generated by the instruction CODE. Other user instructions include constants, variables, and vocabularies.

Instructions are verbs in FORTH language. They are commands given to the computer for execution. Instructions cause the computer to modify memory cells, to move data from one location to the other. Some instructions modify the size and the contents of the data stack. Implicitly using objects on the data stack eliminates nouns in FORTH programs. It is not uncommon to have lines of FORTH text without a single noun. The verbs-only FORTH text earns it the reputation of a "write-only" language.

FORTH is an interpretive language. Instructions given to the computer are generally executed immediately by the interpreter, which can be thought of as the operating system in the FORTH computer. This interpreter is called "text interpreter" or "outer interpreter." A word given to the FORTH computer is first parsed out of the input stream, and the text interpreter searches the dictionary for an instruction with the same name as the word given. If an instruction with matching name is found, it is executed by the text interpreter. The text intepreter also performs the tasks of compiling new user instructions into the dictionary. The process of compiling new instructions is very different from interpreting existing instructions. The text interpreter switches its mode of operation from interpretation to compilation by a group of special instruction called "defining instructions," which perform the functions of language compilers in conventional computers.

Syntax of these defining instructions is more complicated than the normal FORTH syntax because of the special conditions required of the compilation of different types of user instructions. The

syntax of the defining instructions provided by a standard FORTH operating system is summarized in Table II. The most important defining instruction is the ":" or colon instruction. To define colon instructions satisfactorily, a new entity, "structure," must be introduced. This concept and many other aspects involving defining instructions are discussed in the following subsections.

Structures and Colon Instructions

Words are the basic syntactical units in FORTH language. During run-time execution, each word has only one entry point and one exit point. After a word is processed by the interpreter, control returns to the text interpreter to process the next consecutive word. Compilation allows certain words to be executed repeatedly or to be skipped selectively at run-time. A set of instructions, equivalent to compiler directives in conventional programming languages, are used to build small modules to take care of these exceptional cases. These modules are called structures.

A structure is a list of words bounded by a pair of special compiler instructions, such as IF-THEN, BEGIN-UNTIL, or DO-LOOP. A structure, similar to an instruction, has only one entry point and one exit point. Within a structure, however, instruction or word sequence can be conditionally skipped or selectively repeated at runtime. Structures do not have names and they cannot be executed outside of the colon instruction in which it is defined. However, a structure can be given a name and defined as a new user instruction. Structures can be nested, but two structures cannot overlap each other. This would violate the one-entry-one-exit rule for a structure.

Structure is an extension of a word. A structure should be considered as an integral entity like a word inside a colon instruction. Words and structures are the building blocks to create new user instructions from low level to high level. All the instructions created at low levels are available to build new instructions. The resulting instruction set then becomes the solution to the programming problem. This programming process contains naturally all the ingredients of the much touted structure programming and software engineering.

Using the definition of structures, the precise definition of a colon instruction is: *"a named, executable entity equivalent to a list of structures."* When a colon instruction is invoked by the interpreter, the list of structures is executed in the order the structures were laid out in the colon instruction.

When a colon instruction is being compiled, words appearing on the list of structures are compiled into the body of the colon instruction as execution addresses. Thus, a colon instruction is similar to a list of subroutine calls in conventional programming languages. However, only the addresses of the called subroutines are needed in the colon instruction because the CALL statement is implicit. Parameters are passed on the data stack and the argument list is eliminated also. Therefore, the memory overhead for a subroutine call is reduced to a bare minimum of two bytes in FORTH. This justifies the claim that equivalent programs written in FORTH are shorter than those written in assembly language.

Compiler instructions setting up the structures are not directly compiled into the body of colon instructions. Instead, they set up various mechanisms such as conditional tests and branch addresses in the compiled codes so that execution sequence can be directed correctly at runtime. The detailed codes that are compiled are implementation dependent.

Code Instructions

Colon instruction allows a user to extend the FORTH system at a high level. Programs developed using only colon instructions are very tight and memory efficient. These programs are also transportable between different host computers because of the buffering of the FORTH virtual computer. Nevertheless, there is an overhead in execution speed in using colon instructions. Colon instructions are often nested for many levels and the interpreter must go through these nested levels to find executable codes which are defined as code instructions. Typically, the nesting and unnesting of colon instructions (calling and returning) cost about 20% to 30% of execution time. If this execution overhead is too much to be tolerated in a time-critical situation, instructions can be coded in machine codes which will then be executed at the full machine speed. Instructions of this type are created by the CODE instruction, which is equivalent to a machine code assembler in conventional computer systems.

Machine code representation depends on the host computer. Each CPU has its own machine instruction set with its particular code format. The only universal machine code representation is by numbers. To define code instructions in a generalized form suitable for any host computer, only two special compiler instructions, "," (comma), and "C," are needed. "C," takes a byte number and compiles it to the body of the code instruction under construction, and "," takes a 16-bit integer from the data stack and compiles it to the body of the code instruction. An assembly code is thus a number followed by "C," or ",". The body of a code instruction is a list of numbers representing a sequence of machine codes. As the code instruction is invoked by the interpreter, this sequence of machine codes will be executed by the host CPU.

Advanced assemblers have been developed for almost all computers commercially available based on this simple syntax. Most assemblers use names of assembly mnemonics to define a set of assembler instructions which facilitates coding and documenting of the code instructions. The detailed discussion of these advanced instructions is outside the scope of this article.

Constants, Variables, and Vocabulary

The defining instructions CONSTANT and VARIABLE are used to introduce named numbers and named memory addresses to the FORTH system, respectively. After a constant is defined, when the text interpreter encounters its name, the assigned value of this constant is pushed to the data stack. When the interpreter finds the name of a predefined variable, the address of this variable is pushed to the data stack. Actually, the constants defined by CONSTANT and the variables defined by VARIABLE are still verbs in the FORTH language. They instruct the FORTH computer to introduce new data items to the data stack. However, their usage is equivalent to that of numbers, and they are best described as "pseudo-nouns."

Semantically, a constant is equivalent to its preassigned number, and a variable is equivalent to an address in the RAM memory, as shown in Table II.

VOCABULARY creates subgroups of instructions in the dictionary as "vocabularies." When the name of a vocabulary is

called, the vocabulary is made the "context vocabulary," which means it is searched first by the interpreter. Normally the dictionary in a FORTH computer is a linearly linked list of instructions. VOCABULARY creates branches to this trunk dictionary so that the user can specify partial searches of the dictionary. Each branch is characterized by the end of the linked list as a link address. To execute an instruction defined by VOCABULARY is to store this link address into a memory location named CONTEXT. Hereafter, the text interpreter will first search the dictionary starting at this link address in CONTEXT when it receives an instruction from the input stream (e.g., the console terminal).

Instructions defined by VOCABULARY are used to switch context in FORTH. If all instructions were given unique names, the text interpeter would be able to locate them without any ambiguity. The problem arises because the user might want to use the same names for different instructions. This problem is especially acute for single character instructions, which are favored for instructions used very often to reduce the typing chore or to reduce the size of source text. The usable ASCII characters are the limit of choices. Instructions of related functions can be grouped into vocabularies using vocabulary instructions. Context will then be switched conveniently from one vocabulary to another. Instructions with identical names can be used unambiguously if they are placed in different vocabularies.

Creating Defining Instructions

FORTH is an interpretive language with a multitude of interpreters. This is the reason why FORTH can afford to have such a simple syntax structure. An instruction is known to a user only by its name. The interpreter which interprets the instruction is specified by the instruction itself, in its code field which points to an executable machine instruction routine. This executable routine is executed at run-time and it interprets the information contained in the body of the instruction. Instructions created by one defining instruction share the same interpreter. The interpreter which interprets high level colon instructions is called "address interpreter," because a colon instruction is equivalent to a list of addresses. Constants and variables also have their respective interpreters.

A defining instruction must perform two different tasks when it is used to define a new user instruction. To create a new instruction, the defining instruction must compile the new instruction into the dictionary, constructing the name field, link field, and code field – which point to the appropriate interpreter – and the parameter field, which contains pertinent data making up the body of this new instruction. The defining instruction must also contain an interpreter, which will execute the new instruction at runtime. The address of this interpreter is inserted into the code field of all user instructions created by this defining instruction. The defining instruction is a combination of a compiler and an interpreter in conventional programming terminology. A defining instruction constructs new user instructions during compilation and executes the instructions it created at runtime. Because a user instruction uses the code field to point to its interpreter, no explicit syntax rule is necessary for different types of instructions. Each instruction can be called directly by its name. The user does not have to supply any more information except the names, separated by delimiters.

The most exciting feature of FORTH as a programming language is that it not only provides many resident defining instructions as compiler-interpreters, but also supplies the mechanism for the user to define new defining instructions, to generate new classes of instructions or new data structures tailored to specific applications. This unique feature in FORTH amounts to the capability of extending the language by constructing new compilers and new interpreters. Normal programming activity in FORTH is to build new instructions, which is similar to writing programs and program modules in conventional languages. The capability to define new defining instructions is extensibility at a high level in the FORTH language.[4] This unique feature cannot be found in any popular programming language.

There are two methods to define a new defining instruction as shown in Table III. The :–<CREATE–DOES>–; construct creates a defining instruction with an interpreter defined by high level instructions very similar to a structure list in a regular colon definition. The interpreter structure list is put between DOES> and ";". The compilation procedure is contained between <CREATE and DOES>. Since the interpreter will be used to execute all the instructions created by this defining instruction, the interpreter is preferably coded in machine codes to increase execution speed. This is accomplished by the

:–<CREATE–;CODE–

construct. The compilation procedure is specified by instructions between <CREATE and ;CODE. Data following ;CODE are compiled as machine code which will be used as an interpreter when the new instruction defined by this defining instruction is executed at run-time.

Conclusion

Computer programming is a form of art, far from being a discipline of science or engineering. For a specified programming problem, there are essentially an infinite number of solutions, entirely dependent upon the programmer as an artisan. However, we can rate a solution by its correctness, its memory requirement, its execution speed, and other qualities. A solution must be correct. For some applications, the best solution has to be the shortest and fastest. The only way to achieve this goal is to use the computer with an instruction set optimized for the problem. Optimization of the computer hardware is clearly impractical because of the excessive costs. Thus, one would have to compromise by using a fixed, general purpose instruction set offered by a real computer or a language compiler. To solve a problem with a fixed instruction set, one has to write programs to circumvent the shortcomings of the instruction set.

The solution in FORTH is not achieved by writing programs, but by creating a new instruction set in the FORTH virtual computer. The new instruction set in essence becomes "the" solution to the programming problem. This new instruction set can be optimized at various levels for memory space and for execution speed, including hardware optimization. FORTH allows us to surpass the fundamental limitation of any computer, which is the limited and fixed instruction set. This limitation is also shared by conventional programming languages, though at a higher and more abstract level.

FORTH as a programming language allows programmers to be more creative and productive because it enables them to mold a virtual computer with an instruction set best suited for the problems at hand. In this sense, FORTH is a revolutionary development in computer science and technology.

References

1. Moore, C. H., "FORTH: A New Way to Program in Minicomputer," *Astron. Astrophys. Suppl. 15*, pp. 497-511, (1974).

2. "FORTH-79: A Publication of the FORTH Standards Team," FORTH Interest Group, P. O. Box 1105, San Carlos, CA (1980).

3. Main, R. B., "FORTH vs. Assembly," *FORTH Dimensions 1*, 33 (1978).

4. Harris, K., "FORTH Extensibility," *BYTE, 5*, , (1980).

(See pages 28 - 30 for tables)

Formal Definition of FORTH as a Programming Language

(Tables I, II, and III)

Table I

```
LANGUAGE DEFINITION OF FORTH

<character> ::= <ASCII code>
<delimiting character> ::= NUL | CR | SP | <designated character>
<delimiter> ::= <delimiting character> |
        <delimiting character><delimiter>

<word> ::= <instruction> | <number> | <string>
<string> ::= <character> | <character><string>
<number> ::= <integer> | -<integer>
<integer> ::= <digit> | <digit><integer>
<digit> ::= 0 | 1 | 2 | ... | 9 | A | B | ... | <base-1>

<instruction> ::= <standard instruction> | <user instruction>

<standard instruction> ::= <nucleus instruction> |
        <interpreter instruction> |
        <compiler instruction> | <device instruction>

<nucleus instruction> ::= ! | * | */ | */MOD | + | +! | - | -DUP | / |
        /MOD | 0< | 0= | 0> | 1+ | 1- | 2+ | 2- | < | = | > | >R | @ |
        ABS | AND | C! | C@ | CMOVE | D+ | D< | DNEGATE | DROP | DUP |
        EXECUTE | EXIT | FILL | MAX | MIN | MOD | MOVE | NEGATE | NOT | OR |
        OVER | R> | R@ | ROT | SWAP | U* | U/MOD | U< | XOR

<interpreter instruction> ::= # | #> | #S | ' | ( | -TRAILING | . | <# |
        >IN | ? | ABORT | BASE | BLK | CONTEXT | CONVERT | COUNT | CURRENT |
        DECIMAL | EXPECT | FIND | FORTH | HERE | HOLD | NUMBER | PAD |
        QUERY | QUIT | SIGN | SPACE | SPACES | TYPE | U. | WORD

<compiler instruction> ::= +LOOP | , | ." | : | ; | ALLOT | BEGIN |
        COMPILE | CONSTANT | CREATE | DEFINITIONS | DO | DOES> | ELSE |
        FORGET | I | IF | IMMEDIATE | J | LEAVE | LITERAL |
        LOOP | REPEAT | THEN | UNTIL | VARIABLE | VOCABULARY | WHILE |
        [ | [COMPILE] | ]

<device instruction> ::= BLOCK | BUFFER | CR | EMIT | EMPTY-BUFFERS |
        FLUSH | KEY | LIST | LOAD | SCR | UPDATE

<user instruction> ::= <colon instruction> | <code instruction> |
        <constant> | <variable> | <vocabulary>
```

Table II

```
USER INSTRUCTIONS


The statement in parenthesis is according to the FORTH syntax.


COLON INSTRUCTION
<colon instruction> ::= <structure list>
   : <colon instruction>  <structure list>  ;      )

<structure list> ::= <structure><delimiter> |
        <structure><delimiter><structure list>
<structure> ::= <word> | <if-else-then> | <begin-until> |
        <begin-while-repeat> | <do-loop>

<if-else-then> ::= IF<delimiter><structure list>THEN
        IF<delimiter><structure list>ELSE<delimiter><structure list>THEN
<begin-until> ::= BEGIN<delimiter><structure list>UNTIL
<begin-while-repeat> ::=
    BEGIN<delimiter><structure list>WHILE<delimiter><structure list>REPEAT

<do-loop structure> ::= <structure> | I | J | LEAVE
<do-loop structure list> ::= <do-loop structure><delimiter> |
        <do-loop structure><delimiter><do-loop structure list>
<do-loop> ::= DO<delimiter><do-loop structure list>LOOP |
        DO<delimiter><do-loop structure list>+LOOP

CODE INSTRUCTION
<code instruction> ::= <assembly code list>
   CODE  <code instruction>  <assembly code list>     )
<assembly code list> ::= <assembly code><delimiter> |
        <assembly code><delimiter><assembly code list>
<assembly code> ::= <number><delimiter>, | <number><delimiter>C,

CONSTANT INSTRUCTION
<constant> ::= <number>
    <number>  CONSTANT  <constant>          )

VARIABLE INSTRUCTION
<variable> ::= <address>
     VARIABLE  <variable>       )
<address> ::= <integer>

VOCABULARY INSTRUCTION
<context vocabulary> ::= <vocabulary>
     VOCABULARY  <vocabulary>       )
```

(Table III on next page)

Table III

```
CREATING NEW DEFINING INSTRUCTIONS

<high-level defining instruction> ::=
        CREATE<delimiter><compiler structure list>(DOES>)<delimiter>
        <interpreter structure list>;
 : <high-level defining instruction> CREATE <structure list DOES>
        <structure list> ;    )
<low-level defining instruction> ::=
        CREATE<delimiter><compiler structure list>;CODE<delimiter>
        <interpreter assembly code list>
 : <low-level defining instruction> CREATE <structure list> ;CODE
        <interpreter assembly code list>    )
<compiler structure list> ::= <structure list>
<interpreter structure list> ::= <structure list>
<interpreter assembly code list> ::= <assembly code list>
```

DDJ

FORTH 8086 Assembler

(Listing, text begins on page 14)

```
Screen # 6
  0 ( Error messages for 8086 Assembler )
  1 16 bit register not allowed
  2 8 bit register not allowed
  3 address out of range
  4 immediate data value not allowed
  5 missing source register
  6 missing destination register
  7 illegal operation
  8 illegal operand
  9 instruction not implemented
 10 illegal destination register
 11 illegal source register
 12 illegal condition code
 13 register mismatch
 14 destination address missing
 15


Screen # 7
  0 ( 8086 Assembler --- variables )
  1
  2 VOCABULARY ASSEMBLER IMMEDIATE     ASSEMBLER DEFINITIONS HEX
  3
  4 0 VARIABLE       <#>     ( immediate data flag )
  5 0 VARIABLE       <TD>    ( destination addressing type )
  6 0 VARIABLE       <TS>    ( source addressing type )
  7 0 VARIABLE       <RD>    ( destination register )
  8 0 VARIABLE       <RS>    ( source register )
  9 0 VARIABLE       <W>     ( word/byte flag )
 10 0 VARIABLE       <WD>    ( destination word/byte flag )
 11 0 VARIABLE       <OD>    ( destination offset )
 12 0 VARIABLE       <OS>    ( source offset )
 13 0 VARIABLE       <D>     ( direction flag for R/M )
 14 0 VARIABLE       <SP>    ( local stack pointer )
 15 -->


Screen # 8
  0 ( 8086 Assembler --- CASE statement )
  1
  2 ( CASE statement by Charles E. Eaker )
  3 ( from FORTH Dimensions, Volume II, No. 3, page 37 )
  4
  5 : CASE            ?COMP CSP @ !CSP 4 ;            IMMEDIATE
  6 : OF              4 ?PAIRS COMPILE OVER COMPILE =
  7                   COMPILE OBRANCH HERE 0 ,
  8                   COMPILE DROP 5 ;                IMMEDIATE
  9 : ENDOF           5 ?PAIRS COMPILE BRANCH HERE 0 ,
 10                   SWAP 2 [COMPILE] ENDIF 4 ;      IMMEDIATE
 11 : ENDCASE         4 ?PAIRS COMPILE DROP
 12                   BEGIN SP@ CSP @ = 0=
 13                   WHILE 2 [COMPILE] ENDIF
 14                   REPEAT CSP ! ;                  IMMEDIATE
 15 -->
```

(Continued on next page)

FORTH 8086 Assembler

(Listing continued, text begins on page 14)

```
Screen # 9
  0 ( 8086 Assembler --- miscellaneous primitives )
  1
  2 ( save the current stack pointer as base of local stack )
  3 : !<SP>          SP@ <SP> ! ;
  4 ( return number of items on local stack )
  5 : ?<SP>          <SP> @ SP@ - 2 - 2 / ;
  6
  7 ( these redefinitions for error printing prevent )
  8 ( printing of garbage from HERE after assembly has begun )
  9
 10 ( print error message )
 11 : ERROR           "  ? " MESSAGE SP! QUIT ;
 12 ( conditionally print error message )
 13 : ?ERROR         SWAP IF ERROR ELSE DROP ENDIF ;
 14 -->
 15


Screen # 10
  0 ( 8086 Assembler --- destination register processing )
  1
  2 : DREG  <BUILDS C, C, C,          ( store input parameters )
  3         DOES>
  4         DUP C@ DUP OFF =          ( word/byte flag )
  5             IF DROP ELSE DUP <W> ! <WD> ! ENDIF
  6         1+ DUP C@ <TD> !          ( addressing type )
  7         1+ C@ <RD> !              ( set register code )
  8         <#> @ 24 ?ERROR           ( illegal immediate data )
  9         <TD> @ 4 =                ( indexed addressing? )
 10         IF ?<SP> 0 >              ( yes, any displacement? )
 11            IF <OD> ! ENDIF        ( save displacement )
 12         ENDIF
 13         ;
 14 -->
 15


Screen # 11
  0 ( 8086 Assembler --- source register processing )
  1
  2 : SREG  <BUILDS C, C, C,          ( store input parameters )
  3         DOES>
  4         DUP C@ DUP OFF =          ( word/byte flag )
  5             IF DROP ELSE <W> ! ENDIF
  6         1+ DUP C@ <TS> !          ( addressing type )
  7         1+ C@ <RS> !              ( set register code )
  8         <TS> @ 4 =                ( indexed addressing? )
  9         IF ?<SP> 0 >              ( yes, any displacement? )
 10            IF <OS> ! ENDIF        ( save displacement )
 11         ENDIF
 12         ;
 13 -->
 14
 15
```

```
Screen # 12
  0 ( 8086 Assembler --- source register definitions )
  1
  2 ( reg  type      w                 name )
  3
  4 0        2        0        SREG     AL
  5 1        2        0        SREG     CL
  6 2        2        0        SREG     DL
  7 3        2        0        SREG     BL
  8 4        2        0        SREG     AH
  9 5        2        0        SREG     CH
 10 6        2        0        SREG     DH
 11 7        2        0        SREG     BH
 12 -->
 13
 14
 15
```

```
Screen # 13
  0 ( 8086 Assembler --- source register definitions )
  1
  2 ( reg  type      w                 name )
  3
  4 0        3        1        SREG     AX
  5 1        3        1        SREG     CX
  6 2        3        1        SREG     DX
  7 3        3        1        SREG     BX
  8 4        3        1        SREG     SP
  9 5        3        1        SREG     BP
 10 6        3        1        SREG     SI
 11 7        3        1        SREG     DI
 12 -->
 13
 14
 15
```

```
Screen # 14
  0 ( 8086 Assembler --- source register definitions )
  1
  2 ( reg  type      w                 name )
  3 0        4       -1        SREG     [BX+SI]
  4 0        4       -1        SREG     [SI+BX]
  5 1        4       -1        SREG     [BX+DI]
  6 1        4       -1        SREG     [DI+BX]
  7 2        4       -1        SREG     [BP+SI]
  8 2        4       -1        SREG     [SI+BP]
  9 3        4       -1        SREG     [BP+DI]
 10 3        4       -1        SREG     [DI+BP]
 11 4        4       -1        SREG     [SI]
 12 5        4       -1        SREG     [DI]
 13 6        4       -1        SREG     [BP]
 14 7        4       -1        SREG     [BX]
 15 -->
```

(Continued on next page)

FORTH 8086 Assembler

(Listing continued, text begins on page 14)

```
Screen # 15
  0 ( 8086 Assembler --- segment register definitions )
  1
  2 ( reg   type      w                name )
  3
  4 0         5         -1       SREG     ES
  5 1         5         -1       SREG     CS
  6 2         5         -1       SREG     SS
  7 3         5         -1       SREG     DS
  8
  9 0         5         -1       DREG     ES,
 10 1         5         -1       DREG     CS,
 11 2         5         -1       DREG     SS,
 12 3         5         -1       DREG     DS,
 13 -->
 14
 15


Screen # 16
  0 ( 8086 Assembler --- destination register definitions )
  1
  2 ( reg   type      w                name )
  3
  4 0         2         0        DREG     AL,
  5 1         2         0        DREG     CL,
  6 2         2         0        DREG     DL,
  7 3         2         0        DREG     BL,
  8 4         2         0        DREG     AH,
  9 5         2         0        DREG     CH,
 10 6         2         0        DREG     DH,
 11 7         2         0        DREG     BH,
 12 -->
 13
 14
 15


Screen # 17
  0 ( 8086 Assembler --- destination register definitions )
  1
  2 ( reg   type      w                name )
  3
  4 0         3         1        DREG     AX,
  5 1         3         1        DREG     CX,
  6 2         3         1        DREG     DX,
  7 3         3         1        DREG     BX,
  8 4         3         1        DREG     SP,
  9 5         3         1        DREG     BP,
 10 6         3         1        DREG     SI,
 11 7         3         1        DREG     DI,
 12 -->
 13
 14
 15
```

```
Screen # 18
  0 ( 8086 Assembler --- destination register definitions )
  1
  2 ( reg  type      w                 name )
  3 0       4       -1       DREG     [BX+SI],
  4 0       4       -1       DREG     [SI+BX],
  5 1       4       -1       DREG     [BX+DI],
  6 1       4       -1       DREG     [DI+BX],
  7 2       4       -1       DREG     [BP+SI],
  8 2       4       -1       DREG     [SI+BP],
  9 3       4       -1       DREG     [BP+DI],
 10 3       4       -1       DREG     [DI+BP],
 11 4       4       -1       DREG     [SI],
 12 5       4       -1       DREG     [DI],
 13 6       4       -1       DREG     [BP],
 14 7       4       -1       DREG     [BX],
 15 -->
```

```
Screen # 19
  0 ( 8086 Assembler --- miscellaneous operators )
  1 : ?W              <W> @ ;                ( fetch word/byte flag )
  2 : ?TD             <TD> @ ;               ( fetch dest addr type )
  3 : ?TS             <TS> @ ;               ( source addr type )
  4 : ?RD             <RD> @ ;               ( dest. register code )
  5 : ?RS             <RS> @ ;               ( source register code )
  6 : ?OD             <OD> @ ;               ( dest. displacement )
  7 : ?OS             <OS> @ ;               ( source displacement )
  8 : +D              <D> @ 2 * + ;          ( merge direction flag )
  9 : +W              <W> @ + ;              ( merge word/byte flag )
 10 : +RD             <RD> @ + ;             ( merge dest. reg. code )
 11 : +RS             <RS> @ + ;             ( merge source reg. code )
 12 : MOD1            3F AND 040 OR ;        ( set mod field to 01 )
 13 : MOD2            3F AND 080 OR ;        ( set mod field to 10 )
 14 : MOD3            3F AND 0C0 OR ;        ( set mod field to 11 )
 15 -->
```

```
Screen # 20
  0 ( 8086 Assembler --- miscellaneous operators )
  1
  2 ( equates to addressing modes )
  3 0 CONSTANT DIRECT          3 CONSTANT REG16
  4 1 CONSTANT IMMED           4 CONSTANT INDEXED
  5 2 CONSTANT REG8            5 CONSTANT SEGREG
  6
  7 ( initialize all variables and flags )
  8 : RESET          0 <#> !   0 <W> !    0 <OS> !   0 <RD> !
  9                  0 <TD> !  0 <TS> !   0 <OD> !   !<SP>
 10                  0 <D> !   0 <WD> !   0 <RS> !   ;
 11 ( initialize direction pointer for mem/reg instructions )
 12 : DSET           ?TS INDEXED = <D> ! ;
 13 ( set the direction flag true )
 14 : Dt             1 <D> ! ;
 15 -->
```

(Continued on next page)

FORTH 8086 Assembler

(Listing continued, text begins on page 14)

```
Screen # 21
  0 ( 8086 Assembler --- miscellaneous operators )
  1 ( calculate and store relative offset for branches and loops )
  2 ( issue error message if target out of addressing range )
  3 : OFFSET8,   HERE 1+ - DUP ABS 07F > 23 ?ERROR C, ;
  4 ( calculate and store 16 bit offset for JMP and CALL )
  5 : OFFSET16,  HERE  2+  -  ,  ;
  6 ( calculate and store displacement for mem/reg instructions )
  7 : DISP,      <D> @ IF ?OS ELSE ?OD ENDIF DUP  ( get displ. )
  8              IF   DUP ABS 07F >               ( check size )
  9                   IF SWAP MOD2 C, ,           ( two bytes )
 10                   ELSE SWAP MOD1 C, C, ENDIF  ( one byte )
 11              ELSE DROP DUP 7 AND 6 =          ( no displ. )
 12                   IF MOD1 C, 0 C,             ( except [BP] )
 13                   ELSE C, ENDIF               ( always gets )
 14              ENDIF ;                          ( 1 byte )
 15 -->


Screen # 22
  0 ( 8086 Assembler --- instruction processing, groups 1-3 )
  1
  2 ( single byte instructions without operands )
  3 : 1MI   <BUILDS    C,
  4         DOES>      C@ C, RESET ;
  5
  6 ( conditional jumps and loops, 1 byte + offset )
  7 : 2MI   <BUILDS    C,
  8         DOES>      C@ C, OFFSET8, RESET ;
  9
 10 ( string instructions )
 11 : 3MI   <BUILDS    C,
 12         DOES>      C@ +W C, RESET ;
 13 -->
 14
 15


Screen # 23
  0 ( 8086 Assembler --- instruction processing, group 4 )
  1
  2 ( register PUSH and POP )
  3 : 4MI   <BUILDS   C, C,
  4         DOES>
  5         ?TS                                  ( operand type )
  6         CASE
  7         REG16  OF   1+ C@ +RS C,     ENDOF  ( usual 16 bit reg)
  8         SEGREG OF   C@ ?RS 8 * + C, ENDOF   ( segment reg )
  9         28 ERROR                             ( illegal operand )
 10         ENDCASE RESET ;
 11 -->
 12
 13
 14
 15
```

```
Screen # 24
  0 ( 8086 Assembler --- instruction processing, group 5 )
  1
  2 ( instrasegment JMP and CALL, 16 bit offset or reg/mem )
  3 ( intersegment long JMPs and the short JMP are not supported )
  4 : 5MI    <BUILDS  C, C,
  5          DOES>
  6          ?TS                                        ( operand type )
  7          CASE
  8          DIRECT  OF  1+ C@ C, OFFSET16,             ENDOF ( disp 16 )
  9          REG16   OF  OFF C, C@ MOD3 +RS C,          ENDOF ( via reg)
 10          INDEXED OF  DSET OFF C, C@ +RS DISP,       ENDOF ( indexed )
 11          28 ERROR                                   ( illegal operand)
 12          ENDCASE RESET ;
 13 -->
 14
 15


Screen # 25
  0 ( 8086 Assembler --- instruction processing, group 6 )
  1
  2 ( IN and OUT )
  3 : 6MI    <BUILDS C, C,
  4          DOES>                          ( IN or OUT? )
  5          DUP C@ 2 AND                   ( if true is OUT instr )
  6          IF     ?TD ?TS                 ( get operand types )
  7          ELSE   ?TS ?TD                 ( TOS = data width )
  8          ENDIF                          ( NOS = port addr type )
  9          REG16 = <W> !                  ( set word/byte transfer )
 10          REG16 =                        ( check if register addr )
 11          IF     1+ C@ +W C,             ( via DX )
 12          ELSE   C@ +W C, C,             ( via inline port # )
 13          ENDIF RESET ;
 14 -->
 15


Screen # 26
  0 ( 8086 Assembler --- instruction processing, group 7 )
  1
  2 ( ADC, ADD, AND, CMP, OR, SBB, SUB, and XOR )
  3 : 7MI    <BUILDS C, C, C,
  4          DOES>
  5          ?TS IMMED =                    ( source = immediate data )
  6          IF    1+ DUP 1+ C@ +W C,
  7                ?TD CASE                 ( check dest type )
  8                REG8 OF C@ MOD3 +RD C, C,      ENDOF
  9                REG16 OF C@ MOD3 +RD C, ,      ENDOF
 10                INDEXED OF DSET C@ +RD DISP,
 11                     ?W IF , ELSE C, ENDIF     ENDOF
 12                28 ERROR ENDCASE
 13          ELSE DSET C@ +D +W C,          ( source = reg/mem )
 14          ?TD CASE                       ( check dest type )
 15 -->                                                  (Continued on next page)
```

FORTH 8086 Assembler

(Listing continued, text begins on page 14)

```
Screen # 27
  0 ( 8086 Assembler --- instruction processing, group 7 cont. )
  1
  2         REG8 OF ?TS CASE                    ( 8 bit dest. reg )
  3              REG8 OF ?RS 8 * +RD MOD3 C, ENDOF  ( 8 bit source)
  4              INDEXED OF ?RD 8 * +RS DISP, ENDOF ( indexed s. )
  5              28 ERROR ENDCASE                    ENDOF
  6         REG16 OF    ?TS CASE                ( 16 bit dest. reg )
  7              REG16 OF ?RS 8 * +RD MOD3 C, ENDOF ( 16 bit s. )
  8              INDEXED OF ?RD 8 * +RS DISP, ENDOF ( indexed s. )
  9              28 ERROR ENDCASE                    ENDOF
 10         INDEXED OF  ?TS CASE                 ( indexed dest. )
 11              REG8  OF ?RS 8 * +RD DISP, ENDOF  ( 8 bit source )
 12              REG16 OF ?RS 8 * +RD DISP, ENDOF  ( 16 bit source )
 13              28 ERROR ENDCASE                    ENDOF
 14         28 ERROR ENDCASE ENDIF RESET ;
 15 -->


Screen # 28
  0 ( 8086 Assembler --- instruction processing, group 8 )
  1
  2 ( DIV, IDIV, IMUL, MUL, NEG, and NOT )
  3 : 8MI    <BUILDS  C, C,
  4          DOES>
  5          DUP 1+ C@ +W C,
  6          ?TS
  7          CASE                                    ( operand type )
  8          REG8     OF  C@ +RS MOD3 C,     ENDOF   ( 8 bit reg )
  9          REG16    OF  C@ +RS MOD3 C,     ENDOF   ( 16 bit reg )
 10          INDEXED OF  DSET C@ +RS DISP, ENDOF    ( indexed )
 11          28 ERROR  ( direct, immediate, or segment reg illegal )
 12          ENDCASE RESET ;
 13 -->
 14
 15


Screen # 29
  0 ( 8086 Assembler --- instruction processing, group 9 )
  1
  2 ( various shift instructions )
  3 : 9MI    <BUILDS C, C,
  4          DOES>
  5          DUP 1+ C@ <WD> @ +          ( word/byte flag )
  6          ?TS 1 > IF 2 + ENDIF C,     ( shift 1 or count in CL? )
  7          ?TD CASE                    ( operand to be shifted )
  8          REG8     OF     C@ MOD3 +RD C,     ENDOF  ( 8 bit reg )
  9          REG16    OF     C@ MOD3 +RD C,     ENDOF  ( 16 bit reg )
 10          INDEXED OF     DSET C@ +RD DISP, ENDOF   ( indexed )
 11          28 ERROR ( direct, immediate, or segment reg illegal )
 12          ENDCASE
 13          ?TS 2 < IF DROP ENDIF RESET ;    ( clean up stack )
 14 -->
 15
```

```
Screen # 30
  0 ( 8086 Assembler --- instruction processing, group 10 )
  1
  2 ( 2 byte simple instructions without operands )
  3
  4 : 1OMI  <BUILDS   C, C,
  5         DOES>     DUP C@ SWAP 1+ C@ C, C, RESET ;
  6 -->
  7
  8
  9
 10
 11
 12
 13
 14
 15


Screen # 31
  0 ( 8086 Assembler --- instruction processing, group 11 )
  1
  2 ( INC and DEC instructions )
  3   11MI  <BUILDS C, C,
  4         DOES>
  5         ?TS CASE                                  ( operand type )
  6         REG8 OF  OFE C,' 1+ C@ MOD3 +RS C,  ENDOF ( 8 bit reg )
  7         REG16 OF  C@ +RS C,                 ENDOF ( 16 bit reg )
  8         INDEXED OF  DSET OFE +W C,
  9                     1+ C@ +RS DISP,         ENDOF ( indexed )
 10         28 ERROR ( direct, immediate, or segment reg illegal )
 11         ENDCASE RESET ;
 12 -->
 13
 14
 15


Screen # 32
  0 ( 8086 Assembler --- instruction processing, group 12 )
  1 ( MOV instructions reg/reg, reg/mem, or reg/immediate )
  2 : 12MI  <BUILDS
  3         DOES> DROP DSET ?TD
  4         CASE
  5         DIRECT OF  ?TS CASE   ( destination = memory direct )
  6              ( source = 8 bit general register )
  7              REG8 OF ?RS IF 088 C, ?RS 8 * 6 + C, ,    ( reg )
  8                      ELSE 0A2 +W C, , ENDIF ENDOF      ( acc )
  9              ( source = 16 bit general register )
 10              REG16 OF ?RS IF 089 C, ?RS 8 * 6 + C, ,   ( reg )
 11                      ELSE 0A2 +W C, , ENDIF ENDOF      ( acc )
 12              ( source = segment register )
 13              SEGREG OF 08C C, ?RS 8 * 6 + C, , ENDOF
 14              28 ERROR ENDCASE ENDOF
 15 -->
```

(Continued on next page)

FORTH 8086 Assembler

(Listing continued, text begins on page 14)

```
Screen # 33
  0 ( 8086 Assembler --- instruction processing, group 12 cont. )
  1
  2         REG8 OF  ?TS CASE  ( destination = 8 bit general reg )
  3                  ( source = memory direct )
  4                  DIRECT OF ?RD IF 08A C, ?RD 8 * 6 + C, ,
  5                            ELSE 0A0 +W C, , ENDIF ENDOF
  6                  ( source = immediate data )
  7                  IMMED OF 0B0 +RD C, C, ENDOF
  8                  ( source = 8 bit general reg )
  9                  REG8 OF Dt 88 +D C, ?RD 8 * +RS MOD3 C, ENDOF
 10                  REG16 OF 2D ERROR ENDOF
 11                  ( source = indexed address )
 12                  INDEXED OF 88 +D +W C, ?RD 8 * +RS DISP, ENDOF
 13                  28 ERROR ENDCASE ENDOF
 14 -->
 15


Screen # 34
  0 ( 8086 Assembler --- instruction processing, group 12 cont. )
  1        REG16 OF  ?TS CASE ( destination = 16 bit general reg )
  2                  ( source = memory direct )
  3                  DIRECT OF ?RD IF 08B C, ?RD 8 * 6 + C, ,
  4                            ELSE 0A0 +W C, , ENDIF ENDOF
  5                  ( source = immediate data )
  6                  IMMED OF 0B8 +RD C, , ENDOF
  7                  ( source = 16 bit general reg )
  8                  REG16 OF Dt 88 +W +D C,
  9                            ?RD 8 * +RS MOD3 C, ENDOF
 10                  ( source = indexed address )
 11                  INDEXED OF 88 +D +W C, ?RD 8 * +RS DISP, ENDOF
 12                  ( source = segment register )
 13                  SEGREG OF 8C C, ?RS 8 * +RD MOD3 C, ENDOF
 14                  28 ERROR ENDCASE ENDOF
 15 -->


Screen # 35
  0 ( 8086 Assembler --- instruction processing, group 12 cont. )
  1
  2       INDEXED OF ?TS CASE  ( destination = indexed address )
  3                  ( source = immediate data )
  4                  IMMED OF 0C6 +W C, ?RD DISP,
  5                       ?W IF , ELSE C, ENDIF ENDOF
  6                  ( source = 8 bit general reg  )
  7                  REG8 OF 88 +D +W C, ?RS 8 * +RD DISP, ENDOF
  8                  ( source = 16 bit general reg  )
  9                  REG16 OF 88 +D +W C, ?RS 8 * +RD DISP, ENDOF
 10                  ( source = segment register )
 11                  SEGREG OF 08C C, ?RS 8 * +RD DISP, ENDOF
 12                  28 ERROR ENDCASE ENDOF
 13 -->
 14
 15
```

```
Screen # 36
 0 ( 8086 Assembler --- instruction processing, group 12 cont. )
 1
 2       SEGREG OF   ?TS CASE  ( destination = segment register )
 3                   ( source = memory direct )
 4                   DIRECT OF 08E C, ?RD 8 * 6 + C, , ENDOF
 5                   ( source = 16 bit general register )
 6                   REG16 OF 08E C, ?RD 8 * +RS MOD3 C, ENDOF
 7                   ( source = indexed addressing )
 8                   INDEXED OF 08E C, ?RD 8 * +RS DISP, ENDOF
 9                   28 ERROR ENDCASE ENDOF
10
11       28 ERROR    ( illegal destination types )
12       ENDCASE RESET ;
13 -->
14
15


Screen # 37
 0 ( 8086 Assembler --- instruction processing, group 13 )
 1
 2 ( XCHG instructions )
 3 : 13MI  <BUILDS
 4         DOES> DROP DSET ?TD  ( destination type? )
 5         CASE
 6         ( if missing dest reg assume 16 bit reg <-> acc )
 7         DIRECT OF ?TS REG16 =
 8                   IF 90 +RS C, ELSE 28 ERROR ENDIF ENDOF
 9         ( exchange of 8 bit register with reg or memory )
10         REG8 OF 86 +W C, ?TS CASE  ( source type? )
11                   REG8    OF ?RD 8 * +RS MOD3 C, ENDOF
12                   INDEXED OF ?RD 8 * +RS DISP, ENDOF
13                   28 ERROR ENDCASE ENDOF
14 -->
15


Screen # 38
 0 ( 8086 Assembler --- instruction processing, group 13 cont. )
 1
 2         ( exchange 16 bit register with register or memory )
 3         REG16 OF 86 +W C, ?TS CASE
 4                   REG16   OF ?RD 8 * +RS MOD3 C, ENDOF
 5                   INDEXED OF ?RD 8 * +RS DISP, ENDOF
 6                   28 ERROR ENDCASE ENDOF
 7
 8         28 ERROR ENDCASE RESET ;
 9 -->
10
11
12
13
14
15
```

(Continued on next page)

FORTH 8086 Assembler

(Listing continued, text begins on page 14)

```
Screen # 39
  0 ( 8086 Assembler --- instruction processing, group 14 )
  1 ( special routine for TEST instruction )
  2 : 14MI  <BUILDS DOES> DROP ?TS CASE
  3   DIRECT OF 28 ERROR ENDOF          ( direct addr illegal )
  4   IMMED  OF OF6 +W C, ?TD CASE      ( immediate TEST data )
  5          REG8    OF  ?RD MOD3 C,  ?W IF , ELSE C, ENDIF ENDOF
  6          REG16   OF  ?RD MOD3 C,  ?W IF , ELSE C, ENDIF ENDOF
  7          INDEXED OF  ?RD DISP, ?W IF , ELSE C, ENDIF ENDOF
  8          28 ERROR ENDCASE ENDOF
  9   SEGREG OF 28 ERROR ENDOF          ( segment reg illegal )
 10   ?TD REG8 <  ?TD REG16 >  OR  28 ?ERROR  ( other illegals )
 11       84 +W C, ?RD 8 * +RS          ( TEST reg with reg/mem )
 12       ?TS INDEXED < IF MOD3 C,      ( source = general reg )
 13                ELSE DISP, ENDIF     ( source = indexed )
 14       ENDCASE RESET ;
 15 -->


Screen # 40
  0 ( 8086 Assembler --- instruction definitions )
  1
  2 37         1MI     AAA              80 38 38 7MI     CMP
  3 D5 0A      10MI    AAD              A6       3MI     CMPS
  4 D4 0A      10MI    AAM              99       1MI     CWD
  5 3F         1MI     AAS              27       1MI     DAA
  6 80 10 10 7MI     ADC              2F       1MI     DAS
  7 80 00 00 7MI     ADD              08 48    11MI    DEC
  8 80 20 20 7MI     AND              F6 30    8MI     DIV
  9 E8 10      5MI     CALL                              ( ESC )
 10 98         1MI     CBW              F4       1MI     HLT
 11 F8         1MI     CLC              F6 38    8MI     IDIV
 12 FC         1MI     CLD              F6 28    8MI     IMUL
 13 FA         1MI     CLI              EC E4    6MI     IN
 14 F5         1MI     CMC              00 40    11MI    INC
 15 -->


Screen # 41
  0 ( 8086 Assembler --- instruction definitions )
  1
  2                  ( INT )            E9 20    5MI     JMP
  3 CE         1MI     INTO             76       2MI     JNA
  4 CF         1MI     IRET             72       2MI     JNAE
  5 77         2MI     JA               73       2MI     JNB
  6 73         2MI     JAE              77       2MI     JNBE
  7 72         2MI     JB               75       2MI     JNE
  8 76         2MI     JBE              7E       2MI     JNG
  9 E3         2MI     JCXZ             7C       2MI     JNGE
 10 74         2MI     JE               7D       2MI     JNL
 11 7F         2MI     JG               7F       2MI     JNLE
 12 7D         2MI     JGE              71       2MI     JNO
 13 7C         2MI     JL               78       2MI     JNP
 14 7E         2MI     JLE              79       2MI     JNS
 15 -->
```

```
Screen # 42
  0 ( 8086 Assembler --- instruction definitions )
  1
  2 75          2MI       JNZ                 E2          2MI       LOOP
  3 70          2MI       JO                  E1          2MI       LOOPE
  4 7A          2MI       JP                  E0          2MI       LOOPNE
  5 7A          2MI       JPE                 E0          2MI       LOOPNZ
  6 7B          2MI       JPO                 E1          2MI       LOOPZ
  7 78          2MI       JS                              12MI      MOV
  8 74          2MI       JZ                  A4          3MI       MOVS
  9 9F          1MI       LAHF                F6 20       8MI       MUL
 10                    ( LDS )                F6 18       8MI       NEG
 11 8A 8A 8A 7MI         LEA                 90          1MI       NOP
 12                    ( LES )                F6 10       8MI       NOT
 13 F0          1MI       LOCK                80 08 08 7MI          OR
 14 AC          3MI       LODS                EE E6       6MI       OUT
 15 -->

Screen # 43
  0 ( 8086 Assembler --- instruction definitions )
  1
  2 58 07       4MI       POP                 D0 08       9MI       ROR
  3 9D          1MI       POPF                9E          1MI       SAHF
  4 50 06       4MI       PUSH                D0 38       9MI       SAR
  5 9C          1MI       PUSHF               80 18 18 7MI          SBB
  6 D0 10       9MI       RCL                 AE          3MI       SCAS
  7 D0 18       9MI       RCR                                   ( SEG )
  8 F3          1MI       REP                 D0 20       9MI       SAL
  9 F3          1MI       REPE                D0 20       9MI       SHL
 10 F2          1MI       REPNE               D0 28       9MI       SHR
 11 F2          1MI       REPNZ               F9          1MI       STC
 12 F3          1MI       REPZ                FD          1MI       STD
 13 C3          1MI       RET                 FB          1MI       STI
 14 D0 00       9MI       ROL                 AA          3MI       STOS
 15 -->

Screen # 44
  0 ( 8086 Assembler --- instruction definitions )
  1
  2 80 28 28 7MI         SUB
  3             14MI     TEST
  4 9B          1MI       WAIT
  5             13MI     XCHG
  6 D7          1MI       XLAT
  7 80 30 30 7MI         XOR
  8
  9
 10 -->
 11
 12
 13
 14
 15
```

(Continued on next page)

FORTH 8086 Assembler

(Listing continued, text begins on page 14)

```
Screen # 45
  0 ( 8086 Assembler --- control structures )
  1 ( IF...ENDIF or IF...ELSE...ENDIF provides conditional )
  2 ( execution based on the state of the CPU's Z flag )
  3 : IF              074 C, 0 C, HERE RESET ;   ( stores JZ )
  4 : ELSE            0EB C, 0 C, DUP            ( stores JMP )
  5                   HERE SWAP - DUP ABS 07F > 23 ?ERROR
  6                   SWAP 1- C! HERE RESET ;
  7 : ENDIF           DUP HERE SWAP - DUP ABS 07F > 23 ?ERROR
  8                   SWAP 1- C! RESET ;         ( resolves branches )
  9
 10 ( BEGIN...UNTIL construct provides controlled repetitive )
 11 ( execution, exit from loop taken if Z flag is false )
 12 : BEGIN           HERE RESET ;               ( leaves address )
 13 : UNTIL           074 C, HERE 1+ -           ( stores JZ )
 14                   DUP ABS 07F > 23 ?ERROR C, RESET ;
 15 -->


Screen # 46
  0 ( 8086 Assembler --- special operators )
  1
  2 ( force byte addressing mode )
  3 : BYTE            0 <W> ! 0 <WD> ! ;
  4
  5 ( force word addressing mode )
  6 : WORD            1 <W> ! 1 <WD> ! ;
  7
  8 ( set immediate data flag )
  9 : #               1 <#> ! 1 <TS> ! ;
 10
 11 ( comma delimits destination field for direct addressing )
 12 : ,               DEPTH 1 < 2E ?ERROR 0 <TD> ! ;
 13 -->
 14
 15


Screen # 47
  0 ( 8086 Assembler --- loading termination )
  1
  2 ( CODE, ;CODE, and NEXT delimit a code definition )
  3 : NEXT            0E9 C,  NEXT-LINK  OFFSET16,  RESET
  4                   ?EXEC ?CSP SMUDGE [COMPILE] FORTH ; IMMEDIATE
  5
  6 FORTH DEFINITIONS
  7 : ;CODE           ?CSP  COMPILE (;CODE)  [COMPILE] [  ASSEMBLER
  8                   RESET FORTH  [COMPILE] ASSEMBLER ; IMMEDIATE
  9 : CODE            ?EXEC !CSP CREATE ASSEMBLER RESET FORTH
 10                   [COMPILE] ASSEMBLER ; IMMEDIATE
 11
 12 DECIMAL
 13 CR CR  " 8086 Assembler loaded."
 14 CR SP@ DP @ - U. ." bytes left in dictionary." CR CR     ;S
 15
```

End Listing

Excuse Me, What Time Is It?

by Alan Bomberger

Alan Bomberger, 3721 Starr King Circle, Palo Alto, CA 94306.

The article "Time On The Horizon" by D. J. Yates in *DDJ* number 57 stimulated me to explore the timer hardware on the Horizon mother board. Dr. Yates points out that the major weakness of his timing routine is that it fails to keep time without program intervention. This weakness can be overcome by using interrupts.

Interrupts are common in the world of large computers and are, in fact, the heart of multiprogramming systems, for without them, timesharing a computer is difficult. The micro-user, however, rarely timeshares; most systems rely on simple program loops for delays such as are necessary in disk drivers. With no need to keep multiple tasks running simultaneously, delays necessary for floppy disk head movement are most easily accomplished with programmed loops. If accurate timing of external events must be overlapped with computation, this technique breaks down. The method Dr. Yates uses to time events allows for overlap of computation and timing only to the extent that the computations take no more than a single clock period and the clock is sampled continuously by all parts of the system. The method described here allows for consistently accurate timing except through periods of disk access.

I run CP/M on my Horizon and the programs described here are written in FORTRAN-80 and MACRO-80. The use of interrupts and the actual method of using the Horizon hardware clock can be adapted to North Star BASIC or any other language and operating system.

First, a discussion of the use of hardware interrupts on 8080 and Z80 computer systems is in order. The 8080 and Z80 share a common mode of operation that allows for 8 different external interrupts to be processed. When the external interrupt is presented to the computer, one of the 8 RST instructions is forced into the instruction fetch cycle of the computer. The RST instruction is a CALL to a specific location in the computer memory determined by the actual instruction. An RST 4 causes a CALL to location 20H at the end of the currently executing instruction. The code found at location 20H will execute until either another interrupt occurs or an RET instruction is encountered. An RET instruction causes the executing program to resume operation as if nothing has happened except for some time delay determined by the number of instructions in the program started at location 20H. If the interrupt routine changes the contents of any register and does not restore the register, then the program will continue with the new register contents.

The Horizon mother board allows the clock and many other signals to cause any one of the 8 interrupts. Since the Horizon is a Z80 computer, it must be assumed that the Z80 is in interrupt mode 0 (8080 compatibility mode), and it is the responsibility of the programmer to ensure this operating mode. I chose to have the clock flag force an RST 4 instruction for no other reason than 20H was a nice even number (also it did not interfere with any preassigned address in CP/M). I chose a timer interval of 26 ms as a tradeoff between the precise timing available with short intervals and longest maximum period available with a two-byte counter. The choice of 26 ms allows for a maximum software interval of 14 minutes with a two-byte counter (15-bit positive numbers only), where the choice of 3 ms allows for a maximum software interval of 1.5 minutes with a two-byte counter. The use of four-byte integers (allowed in FORTRAN-80) would all but eliminate any need for trade-offs.

Once the interrupt header is properly wired, the execution of an OUT 6 instruction with C0H in the A register will arm the clock interrupt. The first time the clock flag goes high an RST 4 instruction will be forced, the computer will CALL location 20H, and will undoubtedly get stuck in some kind of hideous loop. It is necessary to patch location 20H to jump to some subroutine that preserves the status of all registers and ends with an RET instruction. The construction of the subroutine and the patch of the JMP to it can be accomplished via POKE in BASIC but is done in my example with a FORTRAN callable subroutine.

The subroutine is divided into two parts. The first part is called by FORTRAN and contains two entries. One entry, SETTIM, sets a time interval in a memory location called CLOCK and the other entry, INTVAL, waits for that interval to expire (returning the time that remained at the time it was called). The second part is the actual interrupt routine that executes each time the hardware clock flag goes high and decrements CLOCK once. Its last action before returning is to reset the hardware clock flag so that the routine will be called again at the end of the next interval.

The program does not illustrate the complete range of options available. The programmer may wish to have an entry that returns the time remaining without waiting for the interval to elapse. Also a global "Time of Day" field can be updated at each interrupt. Several interval timers can be decremented to zero at each interval allowing events to be sampled with overlapping intervals. What is important is that the interrupt routine is called each time the clock flag goes high no matter what the main program is doing, as long as interrupts are enabled. The version of CP/M I use has the option of running with interrupts enabled except when the disk drivers are active. These drivers contain programmed loops which are sensitive to timing and cannot be interrupted. The North Star system has a similar user-selected option.

Several comments need to be made about the timer routine presented here. First, this is only an interval timer and the clock interrupts are disabled at the completion of the interval. This helps to ensure that the timer is not left running after the program completes. If the clock interrupt is left enabled after the program that defines the interrupt routine is completed and overlayed by another program, then the next interrupt will cause execution to begin at some random place within the loaded program. This can cause unexpected and disastrous results. If a "Time of Day" clock is to be kept throughout operation of the operating system then the interrupt routine must be loaded somewhere where it won't be overlayed. This could probably be somewhere in high memory or part of the resident system.

Second, note the use of DI and EI instructions both during the setting of the timer and during the interrupt routine. Neither of these is strictly necessary but they protect the code from other interrupts that might fiddle with the clock values. An example where DI is absolutely required is an entry that extends the interval. Such a routine picks up the clock value and adds a new value to the remaining time and returns it. If a clock interrupt occurs during this operation,

the restored value may be one TIC too large, but worse, the interrupt may have reduced the clock value to zero and disabled further interrupts. Thus, the subsequent call to INTVAL will wait for the timer to be reduced to zero but there are no active interrupts to do so. The use of DI and IE around the instructions that add time to the interval prevents such an occurrence.

Finally, note that the timer interrupt routine uses the stack to store registers needed to update the clock. This routine is simple and uses only 6 bytes of stack (the PC on the CALL and the PSW and HL), but a more complicated routine might need to save many more registers. Programs written to run in an environment where interrupts are used must have sufficient stack space to accommodate the largest use of the program and the additional space required by the interrupt routine. It is axiomatic that if an interrupt can be fatal to a program then it will occur at precisely the place the program is vulnerable every time the program is run.

Interrupts can be very . . . Excuse me, What time is it . . . useful, and a mastery of them will lead to a better understanding of multi-user systems.

■■J

(Listing begins on page 52)

What time is it?

(Text begins on page 50)

Listing I

```
      INTEGER  TENSEC
      TENSEC = 10.0/ .026624  +1
      DO 100 I=50,1000,50
      CALL SETTIM(TENSEC)
      DO 50 J=1,I
      B=J/1000.
      A=SIN(B)
50    CONTINUE
      K=INTVAL(TENSEC)
      TIME=(TENSEC - K) * 26.624
      IF(K.EQ.0) GO TO 60
      WRITE(3,200) I,TIME
200   FORMAT(' LOOP OF LENGTH',I5,' TOOK ',F9.3,' MILLISECONDS')
100   CONTINUE
      WRITE(3,201)
201   FORMAT(' ')
      STOP
60    CONTINUE
      WRITE(3,202) I,TIME
202   FORMAT(' LOOP OF LENGTH',I5,' INCOMPLETE IN ',F10.3,' MS',/)
      STOP 'FAIL'
      END
```

Listing II

```
                        .8080
                              ENTRY   SETTIM,INTVAL
                        ;
                        ;     THIS IS A SINGLE CLOCK TIMER ROUTINE THAT IS
                        ;     FORTRAN CALLABLE
                        ;
                        ;     ENTRY TO SETTIM IS WITH THE NUMBER OF TICS
                        ;     TO SET AS AN INTERVAL
                        ;
                        ;     RETURN FROM INTVAL IS AFTER THE TOTAL INTERVAL
                        ;     HAS ELAPSED BUT WITH THE NUMBER OF TICS LEFT IN
                        ;     THE INTERVAL AT THE TIME OF THE CALL AS A FUNCTION
                        ;     VALUE
                        ;
                        ;     THIS ROUTINE USES THE HARDWARE CLOCK ON THE HORIZON
                        ;     MOTHER BOARD IN INTERRUPT MODE SO THAT CONCURRENT
                        ;     COMPUTATION CAN TAKE PLACE DURING THE TIMING INTERVAL
                        ;
0020                    VECTOR  EQU     020H    ; VECTOR ADDRESS FOR RST 4
                        ;
0000'                   SETTIM:
                        ;
                        ;     FORTRAN CALLS SETTIM WITH HL POINTING TO AN INTEGER
                        ;     NUMBER  OF TICS TO SET AS TIMER VALUE
                        ;
0000'  F3                     DI              ; DISABLE INTERRUPTS WHILE FOOLING
                        ;                       WITH THE CLOCK VALUE
0001'  7E                     MOV     A,M     ; GET NUMBER OF TICS
0002'  32 005B'               STA     CLOCK
```

```
0005'  23                      INX    H
0006'  7E                      MOV    A,M
0007'  32 005C'                STA    CLOCK+1 ; SET INTERVAL
000A'  21 0020                 LXI    H,VECTOR; SET UP FOR INTERRUPT
000D'  36 C3                   MVI    M,0C3H  ; A JUMP TO INTERRUPT VECTOR
000F'  21 0035'                LXI    H,TICKER; ADDRESS OF INTERRUPT ROUTINE
0012'  22 0021                 SHLD   VECTOR+1; TO INTERRUPT VECTOR
0015'  3E 50                   MVI    A,050H  ; RESET CLOCK FLAG
0017'  D3 06                   OUT    6       ; ON MOTHER BOARD
0019'  3E C0                   MVI    A,0C0H  ; ARM CLOCK INTERRUPTS
001B'  D3 06                   OUT    6       ; ON MOTHER BOARD
                       .Z80
001D'  ED 46                   IM     0       ; BE SURE IN 8080 MODE
                       .8080
001F'  FB                      EI             ; ENABLE INTERRUPTS
0020'  C9                      RET            ; RETURN TO FORTRAN
                       ;
0021'                  INTVAL:
                       ;
                       ;       WAIT FOR INTERVAL TO ELAPSE
                       ;       RETURN WITH UNEXPIRED NUMBER OF TICS AS FUNCTION VALUE
                       ;
0021'  2A 005B'                LHLD   CLOCK   ; GET CLOCK VALUE LEFT
                       ;
0024'                  INTL1:
0024'  3A 005C'                LDA    CLOCK+1 ; HIGH ORDER NUMBER OF TICS
0027'  FE 00                   CPI    0
0029'  C2 0024'                JNZ    INTL1   ; WAIT
002C'                  INTL2:
002C'  3A 005B'                LDA    CLOCK   ; LOW ORDER AS WELL
002F'  FE 00                   CPI    0
0031'  C2 002C'                JNZ    INTL2
0034'  C9                      RET            ; RETURN WITH UNUSED TICS IN HL
                       ;
0035'                  TICKER:
                       ;
                       ;       ACTUAL INTERRUPT ROUTINE THAT EXECUTES EACH CLOCK TIC
                       ;
0035'  F3                      DI             ; KEEP US SAFE FROM OTHERS
0036'  F5                      PUSH   PSW
0037'  E5                      PUSH   H       ; MUST RESTORE THESE FOR PROGRAM RUNNING
                       ;
0038'  2A 005B'                LHLD   CLOCK   ; GET CURRENT CLOCK
003B'  2B                      DCX    H       ; DECREMENT
003C'  22 005B'                SHLD   CLOCK   ; NEW VALUE - NOT SET CONDITION CODE
003F'  7C                      MOV    A,H     ; HIGH BYTE
0040'  FE 00                   CPI    0       ; IS IT ZERO
0042'  C2 004B'                JNZ    TICTOC  ; NO KEEP TICKING
0045'  7D                      MOV    A,L     ; LOW BYTE
0046'  FE 00                   CPI    0       ; IS IT ZERO
0048'  CA 0053'                JZ     TICOFF ; STOP CLOCK INTERRUPTS
```

(Listing continued on next page)

What time is it?

(Listing continued, text begins on page 50)

```
004B'                           TICTOC:
004B'   3E 50                           MVI     A,050H  ; RESET FLAG
004D'   D3 06                           OUT     6       ; ON MOTHER BOARD
004F'   E1                              POP     H
0050'   F1                              POP     PSW
0051'   FB                              EI              ; RESUME INTERRUPT MODE
0052'   C9                              RET             ; RESUME PROGRAM
0053'                           TICOFF:
0053'   3E 40                           MVI     A,040H  ; DISARM CLOCK INTERRUPTS
0055'   D3 06                           OUT     6       ; ON MOTHER BOARD
0057'   E1                              POP     H
0058'   F1                              POP     PSW     ; RESTORE WORLD
0059'   FB                              EI              ; RESUME INTERRUPTS
005A'   C9                              RET
                                ;
005B'                           CLOCK:  DS      2       ; THE CLOCK WORD
                                        END


Macros:

Symbols:
CLOCK   005B'   INTL1   0024'   INTL2   002C'   INTVAL  0021I'
SETTIM  0000I'  TICKER  0035'   TICOFF  0053'   TICTOC  004B'
VECTOR  0020


No  Fatal error(s)
```

End Listing

Background Information on Text Editors

Text editors, once very vogue systems programming projects, seem to have fallen from grace in recent years. Happily, there are some early signs that this trend is being reversed under the various guises such as "program editors," "structure editors" and "programming environments." I read with interest the recent *DDJ* articles on editors. The authors have several interesting opinions. In order to "stimulate the brilliant – unconventional – hardworking Dr. Dobb's community to start thinking in the direction of what a really good program editor should do," I offer this bibliography on the subject and the sage:

> "In discussing editors firm opinions abound: everyone, from greenhorn to old hand, knows exactly what the best and worst features of given editors are and just how new editors ought to be designed. The only problem is the striking lack of consensus."
>
> Embley, David W., and George Nagy, Behavioral Aspects of Text Editors. *ACM Computing Surveys* 13(1):35, March 1981.

Available References

Barstow, David R., Howard E. Shrobe, and E. Sandewall, Eds., *Interactive Programming Environments.* McGraw-Hill: New York, NY, 1981.

Brown, M. R. and S. R. Wood, *A Display-oriented Program Editor.* Extended Abstract, Yale University, Dept. of Computer Science, New Haven, CT, January 1979.

Card, Stuart, *Studies in the Psychology of Computer Text Editing Systems.* Ph.D. Thesis, Carnegie-Mellon University, Technical Report SSL-78-1, Xerox Palo Alto Research Center, Palo Alto, California, 1978.

Courlouris, F. F., *et al.*, The Design and Implementation of an Interactive Document Editor, *Software Practice and Experience* 6: 271-279 (April-June 1976).

Dam Van, Andreas and D. E. Rice, On-line Text-editing: A survey. *ACM Computing Surveys* 3(3):93-114, Sept. 1971.

Deutsch, P. L., and B. W. Lampson, An Online Editor, *Communications of the ACM* 10(12):793-803, December 1967.

Donzeau-Gouge, V. G., *et al.*, A Structure Oriented Program Editor: A First Step towards Computer Assisted Programming. Technical Report 114, *Iria-Laboria*, France, April, 1975.

Donzeau-Gouge, V. G., *et al.*, Programming Environments Based on Structured Editors: The MENTOR Experience. Technical Report, *Iria-Laboria*, France, April, 1980.

Douglas, Craig C., C – A Guide to the Yale C Editor. *The Quasi-Portable C Editor.* Yale University, Dept. of Computer Science, New Haven, CT, 1980.

Ellis, John R., *MUF: Multiple User Forks.* Technical Report 191, Yale University, Dept. of Computer Science, New Haven, CT, 1981.

Embley, David W., and George Nagy, Behaviorial Aspects of Text Editors. *ACM Computing Surveys* 13(1):33-70, March 1981.

Feiler, Peter H., and Raul Medina-Mora, *An Incremental Programming Environment.* Technical Report, Carnegie-Mellon University, Dept. of Computer Science, Pittsburgh, PA, April 1980.
also in
IEEE Transactions on Software Engineering SE-7(5):472-482, September 1981.

Finseth, Craig A., *Theory and Practice of Text Editors or a Cookbook for an EMACS.* B.S. Thesis Massachusetts Institute of Technology, Technical Report TM-165, Massachusetts Institute of Technology, Cambridge, MA, May, 1980.

Hazel, P., Development of the ZED Text Editor. *Software Practice and Experience* 10(1):57-76, January 1980.

Hunke, H., Ed., *Software Engineering Environments.* North-Holland, 1981.

IEEE Tutorial: *Software Development Environments.* IEEE Computer Society, 1981.

Irons, E. T., and F. M. Djorup, A CRT Editing System. *Communications of the ACM* 15(1):16-20, January, 1972.

Kernighan, B. W., *Advanced Text Editing on UNIX.* Bell Labs Technical Report, Murray Hill, NJ, February, 1979.

Kernighan, Brian W. and P. J. Plauger, *Software Tools.* Addison-Wesley, 1976.

Lampson, Butler, *Bravo Manual.* Xerox Corporation, Palo Alto, California, 1978.

Lewis, J. W., and D. F. Porges, *ALBE/P: A Language-Based Editor for Pascal.* Technical Report, Yale University, Dept. of Computer Science, 1978,
also in
Proceedings of Eighth Texas Conference on Computing Systems, November 1979.

Newman, Ken, *PTED: A Standard Pascal Text Editor Based on the Kernighan and Plauger Design.* Technical Report CSRG-120, University of Toronto, Dept. of Computer Science, October 1980.

Nix, Robert, Experience with a Space Efficient Way to Store a Dictionary. *Communications of the ACM* 24(5):297-298, May 1981.

Peterson, James L., *Design of a Spelling Program: An Experiment in Program Design.* Lecture Notes in Computer Science, Vol. 96, Springer-Verlag, New York, NY, October, 1980.

Peterson, James L., Computer Program for Detecting and Correcting Spelling Errors. *Communications of the ACM* 23(12):676-687, December 1980.

Proceedings of the ACM SIGPLAN/SIGOA Symposium On Text Manipulation. Portland, Oregon, June 8-10, 1981. ACM Order No. 548810, *ACM SIGPLAN Notices* 16(6), June 1981.

Roberts, Teresa L., *Evaluation of Computer Text Editors.* Ph.D. Thesis, Stanford University, November 1979, Technical Re-

by Leonard K. Nicholson

Leonard K. Nicholson, Tulane University, School of Engineering, Department of Computer Science, New Orleans, LA 70118.

port SSL-79-9, Xerox Palo Alto Research Center, Palo Alto, California, 1979.

Samuel, A.,
Essential E. Technical Report STAN-CS-80-796, Stanford University, Dept. of Computer Science, March 1980.

Sandewall, Erik,
Programming in an Interactive Environment: The Lisp Experience. *ACM Computing Surveys* 10(1): 35-71, March 1978.

Shapiro, E. G., *et al.*,
PASES: A Programming Environment for PASCAL. Schlumberger Programming Environment Workshop, June, 1980.
also in
ACM SIGPLAN 16(8):50-57, August, 1981.

Stallman, Richard M.,
EMACS: The Extensible, Customizable Display Editor. Artificial Intelligence Lab Memo 519a, Massachusetts Institute of Technology, Cambridge, MA, 1981.

Teitelbaum, Tim, and Thomas Reps,
The Cornell Program Synthesizer: A Syntax Directed Programming Environment. *Communications of the ACM* 24(9):563-573, September 1981.

Weiner, P., Singh, I., Mostow, D. J., and E. T. Irons, *The Yale Editor E A CRT Based Text Editing System*. Technical Report 19, Yale University, Dept. of Computer Science, New Haven, CT, 1980.

Wood, Steven R., and John R. Ellis,
A Bliss Programming Environment. Technical Report, Yale University, Dept. of Computer Science, New Haven, CT, 1980.

Wood, Steven R.,
Z – The Last Word in a Series of Single Letter Yale Editors. Yale University, Dept. of Computer Science, New Haven, CT, 1980.

Wood, Steven R.,
The Design of a Program Editor: An Alternative. Ph.D. Thesis in preparation, Yale University, Dept. of Computer Science, New Haven, CT, 1981.

Notes

The material from the ACM may be obtained at modest cost from:

Association for Computing Machinery
1133 Avenue of the Americas
New York, NY 10036
Phone: 212/265-6300

The material from Yale University may be obtained by sending a self-addressed, stamped envelope and a nice cover letter to the respecive authors:

Mr. Craig C. Douglas
Yale University
Dept. of Computer Science
2158 Yale Station
10 Hillhouse Ave.
New Haven, CT 06520

Mr. Steven R. Wood
Yale University
Dept. of Computer Science
2158 Yale Station
10 Hillhouse Ave.
New Haven, CT 06520

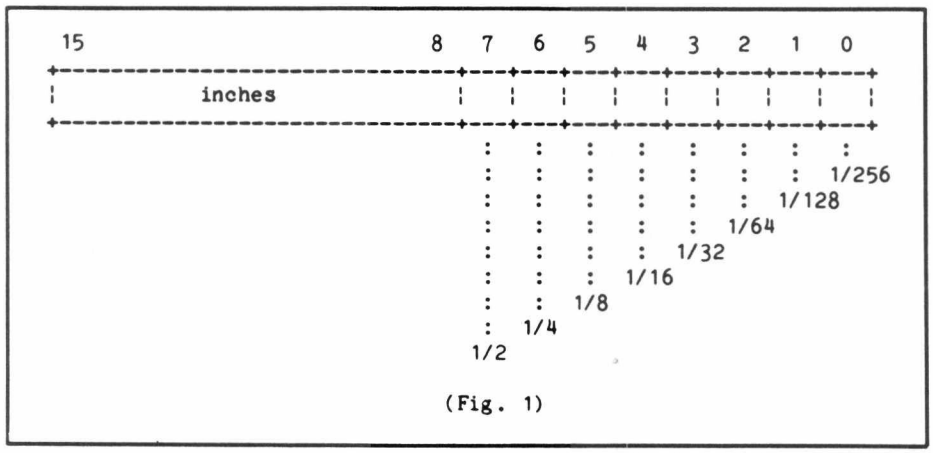
Other technical reports may be obtained from the respective contributors either free or at nominal charge. DDJ

Standards for Computerized Measurement

We should hope that the Metric standard is not adopted in this country because Metric, with its base of ten, aggravates the problems binary computers have always had with floating point numbers. Customary measure is easier to deal with, both for people and computers. It is expected that FORTH enthusiasts will adopt everything that follows without question.

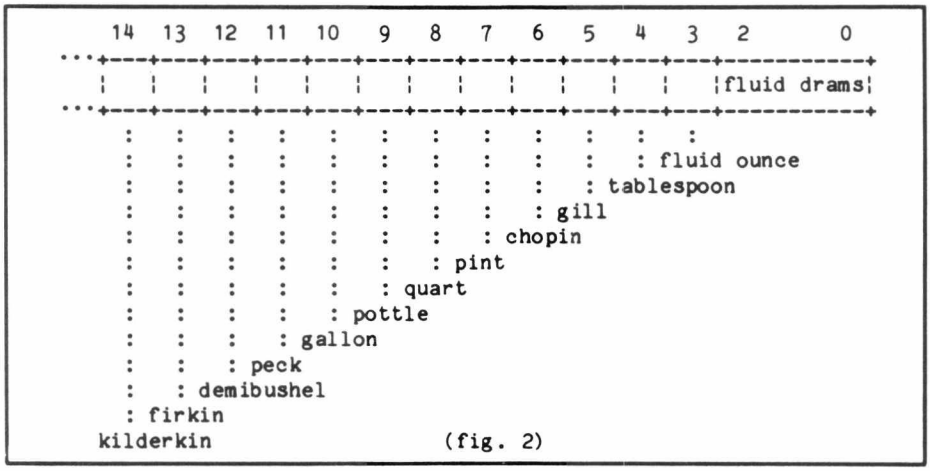
It is easier to divide something (like a stick or a pie) into 16 parts than into 10 parts. That is because dividing by 2 has been well understood for centuries. Dividing by 5 is much more complex.

Consider the inch. On most rulers, inches are broken down into halves, quarters, eights, and so on, depending on the resolution of the ruler. Such fractions are easily represented in a computer and may be acted upon by simple binary arithmetic. Suppose a measurement is to be contained in a word (Figure 1). The high order byte could contain the number of whole inches. The low order byte could contain the fraction of inches, with very fine resolution. We will return to the inch later.



(Fig. 1)

Many of the customary measures show conventional wisdom in the choice of binary units. The most remarkable are the fluid measure (Table 1), which can easily and efficiently be represented in a computer (Figure 2), with varying magnitude and resolution. (Many of the names are archaic. It is left to the reader to decide which are archaic, and which are as up-to-date as tomorrow's newspaper).



(fig. 2)

Many of our standard weights are also based on binary units: an ounce is 16 drams, and a pound is 16 ounces (Table 2). To these I add the Crockford Stone, the Crockford Hundredweight, the Crockford Ton, which is a compromise between the U.S. ton and the Metric ton. Notice that these are scientific units of measure. They are not based on superstitious nonsense like the length of my arm or the diameter of a Krypton atom.

As we have seen, the customary inch is an excellent unit of measure. The Crockford foot is eight inches long, which is good news for foot-long hot dog fans in these days of high meat prices (Table 3). Also, I am defining the Crockford Rod, The Crockford Furlong, and the Crockford Mile, which is a couple thousand inches longer than the old, unscientific mile. It should be just the right length for giving directions: "You go down the road a good Crockford Mile . . ."

Our customary measures of time are based on 60. The Crockford Minute and Crockford Hour correct this oversight, increasing to 64 (Table 4), giving us more time to listen to Andy Rooney. Other new units are the Crockford Day, the Beatle Week, the Crockford Month, and the Crockford Year. These units do not pretend to be based on astrological significance. Standing on the scary brink, as we are, of interplanetary travel, Earth-based time reckoning is unreasonable. Standard Crockford Time will be a boon to business data processors, who have had to work in strange units like Julian dates in order to do time computations.

Finally, in Table 5, we have the Crockford Dollar, which makes truth of the complaint that the dollar is only worth 64 cents (if that). DDJ

by Douglas Crockford

Douglas Crockford, 14541 Countrywood Lane, Irvine, CA 92714.

Customary fluid measures

fluid ounce	=	8 fluid drams
tablespoon	=	2 fluid ounces
gill	=	2 tablespoons
cup or chopin	=	2 gills
pint	=	2 chopins
quart	=	2 pints
pottle	=	2 quarts
gallon	=	2 pottles
peck	=	2 gallons
demibushel	=	2 pecks
bushel or firkin	=	2 demibushels
kilderkin	=	2 firkins
barrel	=	2 kilderkins
hogshead	=	2 barrels
pipe	=	2 hogsheads
tun	=	2 pipes

Table 1.

Crockford weight measures

ounce	=	16 drams
pound	=	16 ounces
Crockford Stone	=	16 pounds
Crockford Hundredweight	=	8 Crockford Stones
Crockford Ton	=	16 Crockford Hundredweights

Table 2.

Crockford linear measures

Crockford Foot	=	8 inches
Crockford Yard	=	4 Crockford Foots
Crockford Rod	=	4 Crockford Yards
Crockford Furlong	=	32 Crockford Rods
Crockford Mile	=	16 Crockford Furlongs

Table 3.

Crockford time measures

Crockford Minute	=	64 seconds
Crockford Hour	=	64 Crockford Minutes
Crockford Day	=	32 Crockford Hours
Beatle Week	=	8 Crockford Days
Crockford Month	=	4 Beatle Weeks
Crockford Year	=	16 Crockford Months

Table 4.

Crockford Monetary measures

Crockford Piece	=	8 cents
Crockford Dollar	=	8 Crockford Pieces
K	=	1024 Crockford Dollars
M	=	1024 K

Table 5.

SOFTWARE REVIEW

ZEN
North Star Management Program
from **Zenrad Controls Company**
1575 A. P. S.
Santa Barbara, CA 93103
Disc and Manual $75, Manual alone $15
Reviewed by **Gene Head**

"ZEN IS HERE . . ."

"ZEN is the most comprehensive management program available for the 8-bit microprocessor of its size." Quoted from the ZEN user's manual.

Well, you can't believe *everything* you read in print. (Not even software reviews.) ". . . of its size", just may be the operative phrase here. ZEN has some good features and is probably a useful tool, but ". . . the most comprehensive . . .", etc., etc. Hardly!

Overview

ZEN is a North Star disk-based program with a screen editor, system monitor, and 8080 assembler. The disk has three copies of ZEN object code (Type 1 files) addressed at 800-H, 8800-H, and A800-H. A version of ZEN compatible with the 100-H North Star DOS (CP/M) is also on the disk. The assembler object file is automatically loaded whenever ZEN operations require assembly. The screen editor and system monitor are contained within ZEN. ZEN will work in as little as 16K but requires 32K to assemble itself. Assemble itself? Yes, both ZEN (the editor/monitor), and ZASM (the assembler) source files are delivered on the disk. This is one of the few really good features of ZEN. The source files, though not well commented, give the user real control. I like that! Alot!

Screen Editor

ZEN does a pretty good job of editing, although I wouldn't call it a screen editor. What you see on the screen is *not* always what you get. In actuality, ZEN uses the North Star line editor with some very good results but clearly not screen editing as I define it.

For instance, as you move the cursor down the screen, everything looks like what you'd expect. Lines are added at the bottom and scroll off the top of the screen. However, moving the cursor up works just fine until the top of the screen is reached. On my twenty-four line display, lines two through twenty-four stay put and line one changes with each cursor movement up. This means when you start back down, all lines above the cursor are in the proper position but lines displayed below the cursor have not been adjusted. Now this is no major problem so long as you are accustomed to it. A few commands to home and clear followed by cursor down movement will work pretty well on shorter files, but forget it on the long ones.

Each line can be edited using the standard North Star edit commands. Here again, this type of editing is pretty primitive. When you insert text into a screen line it doesn't "open up" or expand as new characters are entered. If you forget (me, forget?) to move the cursor to the end of the edited line before a C/R, your line is truncated to the last cursor position. It's exactly like editing programs in North Star BASIC. Very usable in a line-oriented file, but not for screen editing.

ZEN Monitor

There are eleven monitor commands. Standard features include input and output to specific I/O ports, ASCII and hex dumps, jump to, very simple memory test, and hex-to-decimal conversion. The move-memory feature is somewhat different, as overlays are permitted in either direction. This means you can slide any amount of memory forward or backward by any offset value. I'm not certain of the usefulness of this feature but I know my system monitor can't do it!

The READ and WRITE features of the monitor are also questionable when it comes to "Why would you need that feature?" READ will output the contents of memory to a listing device one line at a time. Any keyboard entry except control-C will continue this crude listing feature. The WRITE feature allows absolute keyboard echoing to numbered devices until a control-C is entered to terminate the WRITE function. If there are any North Star users out there without a software-controlled printer-activate function let me know and I'll send you my version of the DOS users area.

Assembler

The assembler can assemble 8080 code without putting any code on disk or into memory. I like this feature, especially when writing user area routines. I can write a modified version of my DOS user keyboard input routine and assemble it without changing memory. If there are no errors and I actually think the code will work, I can then assemble it with a zero offset and it will assemble into RAM. I can supply an offset value that allows source code to be assembled for execution at, say, address 1000-H and store the object code at address 2000-H. ASME will assemble as above but only list any errors encountered.

The LAB command will list all the addresses of labels in the program. Labels are global, as one label buffer is used when assembling more than one file from disk. The LAB command does not show addresses that reference a label.

Up to six breakpoints can be set and registers displayed when the break point address is executed. Continuation after a break is accomplished with the CONT command.

Object code is saved to the object file after exiting ZEN, using the CR and SF commands in North Star DOS.

Conclusion

The manual has a few typos and is pretty confusing in places. I sat down to get a feel for ZEN about five times in a two-month period. It was not the most pleasing experience I've encountered. I had to work at it, not just to learn the syntax and procedures, but to try to figure out what exactly the manual was trying to say.

I think ZEN was developed at a time when North Star needed an editor/assembler. However, with CP/M available, the software houses are providing really quality program development tools at reasonable prices. I would expect ZEN would have been very valuable three or even two years ago, but not today. DDJ

BOOK REVIEWS

The Devil's DP Dictionary
By Stan Kelly-Bootle
Published by **McGraw-Hill**
$7.50, 141 pages
Reviewed by **Joey Robichaux**

I've got this terrible problem.

You see, I'm supposed to review *The Devil's DP Dictionary* by Stan Kelly-Bootle, but . . . well, it's a long story.

The other day I brought the book with me when I went to work. I hoped to read some of it during any (rare) free time I had. When I got to the office, I found out the computer was down because of hardware problems and would be down all day.

Naturally, the programming staff took this as a divine sign to gather and bullshoot in the breakroom all day. Wanting to shoot my share of bull, I entered the break room with my copy of *The Devil's DP Dictionary*.

'What's that you've got there?" popped up almost immediately.

We passed the book around and everyone got a chuckle out of it. The chuckles grew a little louder; of course, more people had to drop in to see what was so funny. This made the chuckles even louder, so even more people came by, which made it even louder, so . . .

Big happy party, right? So where's the problem?

Somehow, during all the confusion and merry-making, I managed to get separated from my copy of *The Devil's DP Dictionary*.

I tried to track it down, though.

Loud, uproarious laughter would spring up down the hall. When I'd get there, I'd find red faces and eyes teary from laughing, but the darned book always seemed one step ahead of me.

So how am I supposed to review this book when it wouldn't stay still long enough for me to finish reading it? I'll try from memory.

I remember a *buffer* was defined as "an old, greasy, and abrasive rag used to clean tape heads."

I remember that BASIC was "originally a simple mid-level language used to test a student's ability to increment line numbers."

I remember that my wife, who knows nothing about computers, didn't think it was funny.

I remember that our programming staff, who also sometimes knows nothing about computers, loved it.

* * * * *

A few days have passed. I've found the elusive copy of *The Devil's DP Dictionary*. It's dogeared, the binding is cracked, the pages are worn from use. It looks like a copy of *Candy* that's been passed around an eighth-grade class.

Our shop certainly enjoyed *The Devil's DP Dictionary*. I hope you do, too.

LISP
By Patrick Henry Winston and Berthold Klaus Paul Horn
Published by **Addison Wesley**
$13.95, 430 pages
Reviewed by **Brian Gordon**

Almost all "serious" intelligent or seemingly intelligent computer programs are written in the same language – LISP. Structurally, LISP is a general purpose list processing language; its data elements are atoms and lists composed of atoms and/or other lists, and its processing power involves the formation of symbolic expressions. As such, LISP is vastly different from FORTRAN, BASIC and other arithmetic-oriented languages more familiar to home computer users; and from Pascal, Forth and other powerful high-level languages popular with the same audience.

Traditionally, LISP has been the language of the artificial intelligence (AI) community, running only on large "mainframe" computers. Despite its potential as a general purpose list processing language, LISP has remained best known as a fairly special-purpose language. It currently has two major dialects, MACLISP and INTERLISP.

Winston & Horn's book, LISP, appears to be aimed at college or graduate students in AI, and concentrates on the MACLISP version of the language, using an appendix to discuss the significant differences in INTERLISP. Since at least one vendor now advertises a micro-based MACLISP, there is reason for interest among home-computer users – and this is an interesting book.

For a reader who met LISP back in the sixties, and who has not seen the changes from the old LISP x.x implementations, the book does a good job of reintroducing the constructs of the language. For someone with no LISP background, but familiarity with one or more other high-level languages, the book may be disconcerting – a lot of material must be covered, and a previous programming background may give you a "mind set" which must be overcome. That is, however, a characteristic of the language, not of the presentation. With no programming background at all, the learning of LISP, while very different from the learning of most other languages, should be no harder. Again, that is a result of the nature of LISP and its most frequent applications areas, not of this specific presentation of it.

Given access to a suitable computer environment, this book should help anyone learn the rudiments of the language. That appears to be half the intent of the text.

The other purposes of the text are many: the teaching of the proper style of LISP programming, some insights into using the language to extend itself, techniques for writing large programming systems, and a fairly large dose of some of the techniques of AI itself. How well each of these objectives is met can best be debated by experts in the various fields involved, but suffice it to say that the conscientious reader will learn the author's views on these and many related topics.

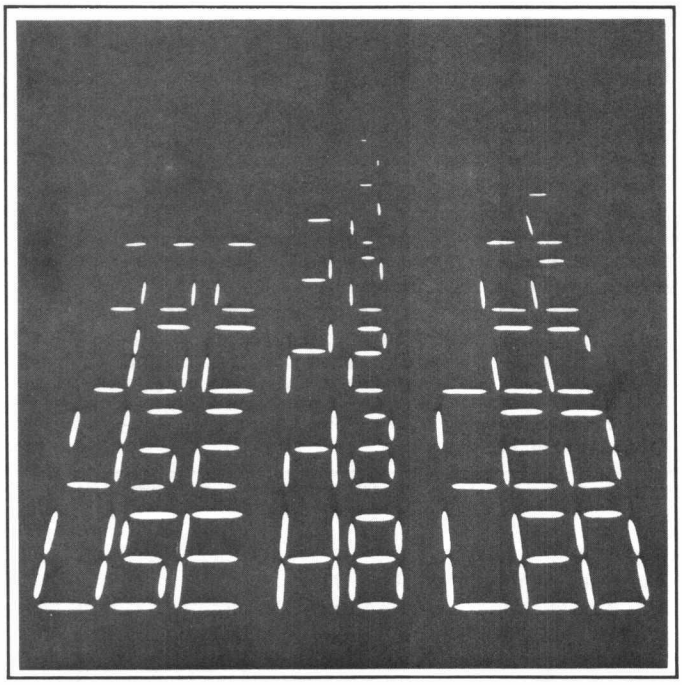
In summary, the book is a readable introduction to at least one dialect of LISP. It contains many examples as well as exercises with solutions, and an extensive bibliography. Its overall usefulness is most limited by your interest in the subject matter, and your access to a machine which speaks MACLISP. DDJ

$2.50
($3.00 in Canada)

DR. DOBB'S JOURNAL

For Users of Small Computer Systems

Number 65 March 1982 Volume 7, Issue 3



Runic: An Interactive, Extensible Compiler
Marty Franz

A Pattern Classification Program
K.F. Wegner and D.E. Raeside

Put the Heath H-8 Front Panel to Work
D.C. Shoemaker

Look inside for even more!

Tightening Leventhal...

...Fast-branch BASIC...

...Pidgin on 8080 & CP/M

DR. DOBB'S JOURNAL
For Users of Small Computer Systems

March 1982 Volume 7, Issue 3, Number 65

CONTENTS

Runic: An Interactive, Extensible Compiler... 12
by Marty Franz

Putting the Heath H-8 Front Panel to Work ... 17
by D. C. Shoemaker

An Improvement Upon a Division Program by Leventhal ... 20
by W. D. Maurer

A Pattern Classification Program ... 23
by K. F. Wegner and D. E. Raeside

Pidgin for the 8080 Under CP/M ... 25
by Andrew L. Bender

Letters ... 4

Dr. Dobb's Clinic ... 6

Of Interest ... 10

Book Review ... 54

Software Review ... 56

Hardware Reviews ... 61

Editor – Marlin Ouverson
Contributing Editors – Dave Caulkins, Dave Cortesi
Marketing Director – Craig Harper
Advertising Sales – Laird Foshay
Circulation – Ari Taylor, Terri Marty
Art Director – Clifford West
Production Manager – Barbara Ruzgerian
Typesetter – Renny Wiggins
Proofreader – Nancy Heubach
Cover Illustration – Al McCahon

© 1981 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, Box E, 1263 El Camino Real, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025.

EDITORIAL

Have you ever watched synchronicity? Editors have to. In the absence of real geniuses (genii?) it seems everyone ends up doing roughly the same things at roughly the same time. Variations on themes which are just other variations on other themes... But as Heinlein noted in his great Foundation trilogy, mutants are unpredictable. They defy synchronicity except when it pleases them. Think of "mutants" as free-thinkers, unbounded by mediocrity or convention.

We had already known this would be the year of synchronous announcements about hard-disk interfaces for micros. And some 16-bit machines made their expected appearances at the top end of the market. Rumors of an Atari to include dual 68000's presages a 16-bit consumer/hobbyist micro, as did the gossip mongers from Apple and Japan. But all was upstaged by Radio Shack's announcement in January of their Model 16, with an MC68000 cpu. What, you ask, only one? Well, they threw in a Z80 to handle mundane things like I/O.

We will bring you more on the machine as soon as we get our eager hands on one. For your information, it costs $4999 if you need only one disk drive (after all, these disks hold 1¼ Megs). And the TRS-Folks amiably allow their new pet the run of all Mod 2 software. Just an added convenience.

My, how times have changed! *DDJ* got its start trying to fit a tiny version of Basic into a mere handful of bytes (do 6K fit in one hand or two?) and here we are, almost casually mentioning a micro which upgrades to 512K RAM (a slight 128K is standard) and supports multi-users.

I wonder what'll happen next month?

Marlin Ouverson – Editor

Subscription Rates: $25 per year within the United States; $40 for first class to Canada and Mexico; $58 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

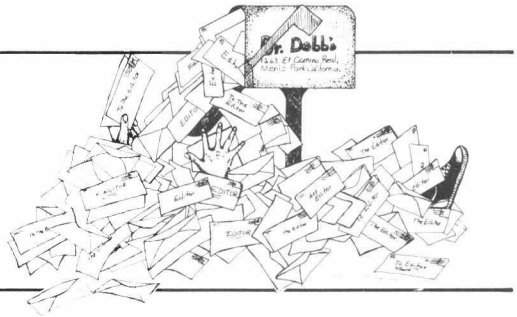
Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Eric H. Johnson, Jerry Horn, Edward W. Carr, Steven Fisher, Thomas H. Grohne, G. V. Elkins, Robert M. Connors, William J. McEown, Robert C. Luckey, Kim Leary, Robert N. Stabler, DeWitt S. Brown, John B. Palmer, E. Glion Curtis, Jack Miller, Burks A. Smith, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Howard E. Decker, Jim Wilson, Frank Lawyer, Chris Pettus. **Retaining Subscribers:** Osamu Shimizu, Scott B. Guthery, Robert L. Edens, Titus Purdin, Steven Fisher Enterprises. **Sustaining Subscribers:** Gordon Eubanks, Jr. and Pete Roberts. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** LP Enterprises, 8/11 Cambridge House, Cambridge Road, Barking, Essex, 1G11 8NT, England (Tel. 01-591-6511, TLX 892395). Hofacker-Verlag, Tegernseer Strass 18, D-8150 Holzkirchen, West Germany. **Asia & Austraila:** Computer Store, POB 31-261, 22B Milford Road, Milford, Auckland 9, New Zealand. ASCII Publishing Co., Ltd., Aoyama Building 5F, 5-16-1 Minami Aoyama, Minato-ku, Tokyo 107, Japan.

LETTERS



He Picks IBM's Nits

Dear Doctor Dobb:

I have bought an IBM Personal Computer and discovered a few annoying items.

Nit #1: the system demands a date on power-up (with DOS). This I do not like (right now), but I do not want to subterfuge it because *someone* hopefully will make a CMOS date/time clock board with software to load those features automatically on power-up.

Nasty Nit #2: FORMAT program will format virtually any disk, no matter how bad. It gives no information while running (like what track it is on, is it 'F'ormatting, 'E'rror, or 'V'erified). Disks with large scratches and badly warped surfaces are blithely accepted with absolutely no comment. The display is no different with good diskettes.

Related Nasty Nit #3: CHKDSK tells only how many files there are, and by a little calculation, one can discover if a file BADTRACK was written. Beyond this, the error information is absent. A formatted display of the error info in BADTRACK would be helpful here, as well as when FORMAT is complete.

Nit #4: There is absolutely no way to use CHKDSK on other diskettes, in a single drive system, despite instructions to the contrary.

Nit #5: The monochrome display/printer adapter card has un-documented features:

a) J1 (2-pin connector)
   open (normal) = normal operation
   shorted (test??) = disable monochrome display. Horizontal drive gets killed, but the schematic shows vertical drive being disabled, along with underline being toggled.

b) J2 (6-pin connector)
   - This is near the 9-pin DB connector on the adaptor card, and is for a light pen.
   - The schematic is incomplete on page D-20 (Technical Reference Manual): it does not indicate that pin 1 goes to "+ LPEN (SHT 6)".
   - For positional & pin documentation, see color graphics monitor adaptor, page 2-63.
   - For I/O register programming data, see page 2-63 and assume port address x '3BB'. Using DEBUG to address either port 389 or 3BB resulted in a x 'FF', no matter what state (open, ground or +5v) pins 1 and 3 on J2 were set to. So either my board is bad, or the hardware on J2 is window dressing.

Nit #6: I realize that up to 256K of RAM can be put inside the system unit (Godbout, *et al.* . . . How about a 256K memory board for us?? Later, a 512K board??). And that 1 Megabyte is addressable.

BUT, there is no excuse for (a) 48K of ROM, where only 8K is available for the user (read 40K of SYSTEM ROM!!), and (b) the ROM routines mostly are not written as subroutines, so many otherwise useful routines need to be duplicated elsewhere.

OVERBYTE GALORE!

Nit #7: there is no software/hardware interface for Winchester drives, or for that matter, for 8-inch floppy(ies). A big machine like this needs mass storage (5 Megabytes or more) for reasonable business (and some hobby) applications.

Please help!

Yours,
"Polarbaer on an Ice Flow"
Robert Baer
921 Lincoln Ave.
Palo Alto, CA 94301

Getting Hyper?

Dear Marlin:

William Brogden's guest editorial about program editors in the June issue of *Dr. Dobb's* definitely got my attention and prompted this letter. I have been informally studying various editors off and on for the last few months. I intend to use what I learn to improve the editor on my own system. My first ideas for improvement came from my own experiences since I have used many different editors at work to develop engineering applications programs. My desired improvements, however, quickly got out of hand so I turned to textbooks and computer science journal articles.

The article "On-line text editing: a survey," by A. van Dam and D. E. Rice in the September 1971 *Computer Surveys* gives a thorough description of several different editing systems. Three of these systems are very much like the system sketched out by Brogden. Brown University developed HES and FREES while the Augmented Human Intellect Research Center at Stanford Research Center developed the NLS system. These systems allow the user to link text areas together to form a cross-referenced "hypertext." (HES stands for Hypertext Editing System.) This cross-reference system can be used to display the hierarchy of the program in a two-dimensional manner under the control of the user.

My descriptions of these systems are very sketchy and incomplete since I am writing from brief notes about a general review article. I also have not covered any recent work since I could not readily find a more recent review article. This letter merely points the way to past work on advanced text editing systems done at Brown University and Stanford Research Center. This past work would provide a strong foundation on which Brogden's advanced text editor could be built.

Sincerely,
G. Wayne Smith
7413 S. Woodward
Woodridge, IL 60517

Editor's Note: Anyone interested in some creative thought on text editing should read Literary Machines, *by Ted Nelson. Send $15 to him at Box 128, Swarthmore, PA 19081.*

TEKnicalities

Dear Mr. Ouverson:

Here are some notes concerning my article, "TEK4010: A Tektronix Simulator," *DDJ* #60, October 1981. First, in the introductory paragraph, the line "or perhaps C) could be directly on" should include "run" after "be." Second, on page 10 paragraph 1 (9th line), 0F00H should read 0F000H. There was also a slight problem which occurred in a machine-dependent part of the program listing. Apparently, one line in a subroutine SIOISR was lost when the source code was transferred from one computer to another for printing. The fourth instruction SIOISR should be "INP A". This should cause no trouble to this implementing TEK4010 since it is in the machine dependent code section. However, anyone who studies this routine for the purpose of producing their own circular input buffering would have been confused by the lack of this instruction.

As a final note, I should also like to mention that the DIV8 routine which does 16-by-8-bit division is by no means optimal. I recommend use of a more standard type algorithm if this routine proves too slow for a given implementation of TEK 4010. *DDJ*'s "Clinic" has discussed division routines on several occasions including the October issue where TEK4010 appeared.

I would enjoy hearing from anyone who works with TEK4010 or has any reaction to this software package.

Sincerely,
Anthony Skjellum
1695 Shenandoah Road
San Marino, California 91108

Tutti-Frutti AI

Dear *DDJ*:

Many thanks for the introduction of the subject of LISP and some allusions to artificial intelligence, in the form of Ronald Nicol's December issue piece, "Actors, Property Lists, and LISP." We have a long battle ahead of us to convince the public that AI, far from evil magic, is simply another method of processing orientation, data-directed, which permits a computer to acquire greater decision-making independence. As such, AI techniques of many flavors have already left the realm of this computer science epithet to settle in as viable real-world application tools. All are encouraged to familiarize themselves with the difference between procedural languages, and list-processing languages; but more so, on the necessity to build any software around carefully planned data structures.

As one who struggled horribly with Winston, feeling stupid at her inability to comprehend his jargon, despite the aid of attendance in a very clear seminar on the very same historic summary of AI, given at Carnegie-Mellon; and had worse beginner's trembles wrestling through Nilsson, and CMU's own Newell and Simon, at the very time these august gentlemen lectured live in front of her classroom seat twice weekly; I must object to the bibliography appended to the article. These references will likely turn off the beginning researcher, discouraging further attempts to abandon old computing methodologies in favor of AI types. (Incidently, Nilsson, also very lucid in person, comments that AI cannot currently be fixed in definition, so that our coming students will still wander freely with their own experimentations, expanding this fledgling science. Nilsson's popularly shared opinion has prevented the preparation of a comprehensive beginning text or lay source in AI.)

The article itself aims at *Dr. Dobb's* highly sophisticated audience, and seems difficult for a fairly new computerist to absorb, though list and AI concepts are not. Nor, of course, is it complete showing primarily the property list, and a few other basic LISP functions. There are many other approaches which bypass property listing all together. But magazine pieces are, by necessity focused to narrow subject matter.

So where does the interested reader go to expand his awareness? First, avoid Siklossy. He is outdated, inaccurate, hard-to-read. Postpone Winston, or you may not figure out that CONS = IF . . . THEN, since all is clothed in friendly, but high-falutin' academic style. The best start-up I know is a darling, public-oriented, slightly-old paperback by Bertram Raphael entitled *The Thinking Computer (Mind Inside Matter.)* San Francisco: W. H. Freeman and Co., 1978. Second, if you have access to a video player, and are willing to splurge or chip in with some friends, consider renting the tv Nova series episode, *Mind Machines*, from Time Life Video. This presentation of AI conducted by the originators themselves, makes an excellent program for computer club or school audience.

Unfortunately, the only tutorial on LISP for beginners I know about, an extremely well-done series of lessons prepared and tested in a live classroom, remains as yet unpublished. But if you are willing to do a little work to translate DEC20-oriented MacLisp to your version, and would like to learn much more about AI, predicate calculus, recursion, (explained by a dragon), Turing, and much more, program a robot to wander about a house, and generally ape all the

(Continued on page 22)

Ignore this message !

What message?

Dear *DDJ*,

I read with interest the articles on printer drivers for the MX-80 printer in the December issue. My problem is that I do not have room in my BIOS for more than a very short printer driver so I have had to use an alternate approach to insert printer control codes into the text.

Some of the control codes for the printer are used for other purposes by CP/M or by ED and may not be sent to the printer driver. Some of the printer control codes are preceded by an ESCape, which in my case is a problem because ESCape is used to leave a terminal emulator to return to the system.

This problem also occurs for the Centronix printers. R. L. Brechwald of our department had developed a procedure for inserting printer control codes into text that I have found to be a satisfactory solution. The procedure is to develop a set of library files that can be used with ED to insert code to turn on or off various features of the printer when the text is printed. The main drawback of this approach is that the library files consist of a few bytes of code, but use up a minimum of 1K of disk space. The S mode of DDT of SID can be used to create such files. For example, an SO or 0EH followed by 1AH, an end of file, can be saved as SO.LIB. This can be inserted with the R command of ED to turn on the double width letter mode of the printer. DC4.LIB can contain the code to turn off the double width mode.

In the above example, an enhanced double-printed, double-width line will be printed followed by a line printed in compressed text. The example demonstrates that this is not an elegant solution to the problem but it does allow one to use all of the printer's features.

John S. Neff
Professor of Astronomy
University of Iowa
Iowa City, IA 52242

```
?
A>ED EXAMPLE.TXT
NEW FILE
     : *#AV
     : *RFF
     : *RSO
     : *RESCE
     : *RESCG
     : *i
    1:          Ignore this message !
    2:
     : *rdc4
     : *rescf
     : *resch
     : *i
    2:
    3:
     : *
     : *rsi
     : *i
    3:          What message?
    4:
     : *rdc2
     : *rff
     : *b22t
    1:  ^L^N^[E^[G      Ignore this message !
    2:  ^T^[F^[H
    3:  ^O       What message?
    4:  ^R^L*
    2:  ^T^[F^[H
    2: *e
A>TYPE EXAMPLE.TXT
```

DR. DOBB'S CLINIC

by D. E. Cortesi, Resident Intern

It Ain't Necessarily So

Here's a true, sad story. It seems that one of the first APL interpreters ever built suffered a bit of overhead in passing from one line of program text to the next. Word got around in the tiny community of users that one way to speed up a program was to cram as much onto one line of code as possible, rather than spreading the logic out over several lines. Those users became the gurus of the next generation of users, and they of the next, and so on. The bit of folklore was passed along and accepted uncritically by all who heard it. It's easy, and wickedly enjoyable, to cram a lot into one line of APL code. It's still done today, even though no interpreter built in the last ten years has imposed any significant penalty on a multi-line program. Those who use the language think (without verifying it) that they are optimizing their code; those who don't use the language find the one-liners impossible to read.

Something similar is happening today with BASIC. The interpreters written for today's personal machines apparently impose a time penalty on program branches, a penalty proportional to the position of the target line in the program. Hobbyists and hackers discover this, and pass the information on to others. "Put critical subroutines at the top of the program," the folklore runs. "Put program initialization code at the end; the main code will run faster." "A delay loop has to be a one-liner, or the delay will vary depending on the line number." And so on.

All this is thoroughly bad advice. Not necessarily *false* advice, but *bad* advice. Programs are distorted into unreadable shapes in order to compensate for the failings of the language implementation. Those failings could be corrected easily, and probably will be corrected in later interpreters. But the folklore will go on, long after the system builders remove its rationale.

In this column we will review several different ways in which a BASIC interpreter could implement branches (GOTOs and GOSUBs). We'll see how ordinary techniques of systems programming can reduce the overhead cost of a branch to a trivial level and make it independent of the target line's position.

Defining Terms

First we must set up some terminology. A BASIC interpreter works on a *program text* which is composed of *lines*. The lines are adjacent in storage. We'll assume a line consists of a sequence of bytes,

NN X . . . X T

where NN is a 16-bit integer, the line's *number*. The variable number of bytes X . . . X are the text of the line, and the byte T is a unique terminator that marks the end of one line and the start of the next. There are n lines in the program text; they have an average length of b bytes.

We'll discuss algorithms in pseudo-code; in the pseudo-code all variables represent 16-bit values, declared as integers or as addresses. We'll assume that the interpreter keeps two global addresses A and Z. A points to the first line; Z points to a dummy line that follows the last line of the program text. Another global address, P, points to the current line.

The function NUM(X) denotes the number of the line addressed by X. SCAN(X) denotes advancing X to the succeeding line by scanning one byte at a time to the byte after a terminator.

The problem we want to solve is this: given a target line number T, assign a value to P so that T=NUM(P).

Simple-Minded Method 0

Method 0 is the method that people attribute to Microsoft BASIC for the TRS-80 and to the Apple and Atari BASICs. Its logic is painfully simple (Figure 1). In order to analyze it we must make some assumptions about the distribution of the target lines the program will reference.

We'll make a simple (and not very realistic) assumption that T is a uniform, random selection from all the line numbers in the program. Then a call of Method 0 will on the average invoke SCAN(X) half as many times as there are lines in the program. The further the target line is from the start of the text, the more times SCAN will be called, and the longer the branch will take.

Mediocre Method 1

A ridiculously simple improvement can be made on Method 0, an improvement that is so trivial it is hard to see why it isn't in use now. All that is required is that the interpreter distinguish between forward and backward branches (Figure 2). If the target line number T is less than NUM(P) then the search should start at the top of the text and proceed to P. If T is equal to or greater than NUM(P) then the scan should start at P and proceed to Z. Note that this logic defines a branch from line P to line P as a forward branch, so that when T=NUM(P) initially, the routine takes minimum time.

If we assume that P and T are independent – almost certainly not the case – then Method 1 scans a partition of the program text that is, on average, $n/2$ in size. Since T is assumed to be uniformly distributed, Method 1 will find it half way through that partition, looking at $n/4$ lines, or half as many as Method 0. We can do better.

Somewhat Brighter Method 2

Suppose that, as part of executing the RUN command, the interpreter whips through the program text and sets M, the address of the line that begins nearest to the midpoint of the text. That wouldn't cost much time, and it only has to be done once when the program is started. Then the branch logic of Method 2 can be implemented (Figure 3).

Method 2 is an extension of Method 1; it chooses one of six overlapping partitions of the program text to scan. It can never scan more than $n/2$ lines. Under the assumptions that P and T are independent and uniformly distributed, we expect it to look at $n/6$ lines. That's a 50% improvement on Method 1 and three times as fast as Method 0. We can do better.

Methods 3, 4, and 5

The fundamental problem with all these methods of branching is that they rely on the procedure SCAN, a byte-by-byte scan over a line. SCAN must take time proportional to the average line-length, b. Method 0 takes time proportional to $bn/2$, Method 2 time proportional to $bn/6$. Now, b is a fairly large number; 40, say, in a typical program and 120 or more in some very dense games. We can get real speed by doing away with the factor b.

That can be done at a storage cost of two bytes per line. Re-specify the form of a line to be

NN LL X . . . X T

where LL is a length word that gives the distance in bytes to the start of the next line. A BASIC interpreter is in part a text editor, and it might be awkward to have

```
Method_0:

    global addresses A, P, Z
    local address S

    if ( T=NUM(P) ) then exit.

    S := A
    while ( NUM(S)<>T ) and ( S<Z ) do SCAN(S)

    if ( S=Z ) then ERROR
               else P := S

end Method_0.
```

Figure 1

```
Method_1:

    global addresses A, P, Z
    local addresses S, Q

    if ( T<NUM(P) ) then     { backward branch }
        S := A ; Q := P
    else                     { forward branch }
        S := P ; Q := Z
    endif

    while ( NUM(S)<>T ) and ( S<Q ) do SCAN(S)

    if ( NUM(S)<>T ) then ERROR
                     else P := S

end Method_1.
```

Figure 2

to maintain LL while editing. That doesn't matter. The editor need only allow for the extra two bytes. When the RUN command is given, the interpreter can pass once over the program text to set the correct length in each LL word. If the editor will be kind enough to set LL negative when a line is changed, this pass can be made even more quickly.

With LL available, we can replace SCAN(X) with SUCC(X), a procedure to increase the contents of X by the value of LL in the line it points to. Let's suppose that SUCC takes about the same time as a SCAN of a 4-byte line. Then using SUCC means that the factor b has been replaced wih a constant of 4. Methods 3, 4, and 5 are simply Methods 0, 1, and 2 with SUCC replacing SCAN. Method 0 takes time proportional to $bn/2$; Method 5 takes time proportional to $4n/6$. If b is around 40, then Method 5 runs in about 3% of the time of Method 0, or 30 times faster. We can do better.

Method 6

The procedure SUCC has to perform an add in order to develop the address of the next line. Why not keep the actual address in the LL word instead? The LL word probably should not contain an address at all times. When the program text is written to a file, lengths would be more appropriate. And addresses complicate matters even more for the editing phase.

Addresses are needed only during execution; that is, from RUN time to END time. Suppose that, at RUN time, the interpreter passed over the text and set all LL fields to contain the address of the next line (that might even be quicker than setting lengths). Then at END time it could pass over the text again and replace the addresses with lengths.

Now the operation SUCC(X) becomes a simple 16-bit load. That can be assumed to run in half the time of SUCC(X) when lengths are used. Method 6, then, is Method 5 with this change made; under the assumptions we've been using, it would use less than 2% of the time of Method 0. A branch still takes an amount of time relative to the position of the target line, but the absolute numbers are getting very small. However, we can do better.

Smart Method 7

So far we have made smaller partitions of the text and scanned them faster. One bottleneck remains: in our search for line T we make a number of comparisons where $T<>$NUM(S). If we could reduce the number of comparisons, the code would go faster. In order to do so, we have to eliminate sequential searching.

Remove the LL words from the lines. Instead, keep a global count of the n number of lines in the program text. At RUN time, allocate n words of free storage and call it vector V. Pass over the program text looking for terminator bytes, and set every V_i to be the address of the ith line of the program. The storage cost, note, is still two bytes per line.

Define new variables a, z, and p such that $V_a=A$, $V_z=Z$, and $V_p=P$. Now the problem of finding a branch target becomes one of finding i such that NUM(V_i)=T, and then setting $p=i$ and $P=V_i$.

This can be done with a standard binary search, which we can expect will iterate about log2(n) times. A pseudocode sketch of such a search looked as if one iteration of the search loop was about equal in complexity to four iterations of the Method 6 while-loop.

Suppose we had a 240-line program. We would expect Method 7's loop to iterate about log2(240) or 8 times. Method 6's while-loop would iterate roughly $n/6$, or 40, times. Method 7, then, has an average expected time cost about 80% of Method 6's.

More important, the time taken by Method 7 is independent of the location of the target line – it depends only on n, the number of lines. This removes the underpinnings of the folk-wisdom on organizing programs. The initialization code and explanatory comments can go at the top, the subroutines at the bottom, and no time penalties will be incurred. This advantage (as well as a significant overall speed-up in execution) is achieved at a storage cost less than 10% of the size of the program text, given typical line-lengths, and a small overhead cost at RUN time.

Method 8

If we are willing to pay a cost of three bytes per line, we can do better still. At RUN time, set aside $1.5n$ words of storage in a table H, and zero it. Then pass over the text and form a hash code from each line number. Use the hash code as an index to store the address of the line in H.

With a table of size $1.5n$, and assuming a decent hash function, there should be few collisions, and they can be dealt in a number of standard ways. A conservative estimate would be that any target line T could be found in 1.5 probes of the table, on the average. The performance of Method 8 would depend on the speed of the hashing function. In a machine that has an integer divide instruction in hardware, Method 8 should be considerably faster than Method 7, with the result that a branch to any line could be executed in a time that is just not significant. Furthermore, the minuscule time penalty would be a constant, independent of program size.

Conclusion

There are well-known ways in which a BASIC interpreter could execute branches in a trivial amount of time, and independently of the location of the target line. Methods 7 and 8 require extra data space amounting to less than 10% of the program size, and at most a few hundred bytes of interpreter code. Method 7 is most suitable for 8-bit CPUs, Method 8 for machines like the 8086, that have hardware divide. With 64K becoming a standard size and larger storage appearing frequently, the extra space taken by these methods will become unimportant in a very few years. Even now, an interpreter could start out using Method 7, then drop back to Method 2 if free storage ran short.

We can expect, then, that the conventional wisdom on how to organize a BASIC program for speed will lose all basis in fact in a fairly short time. Writers who advocate such conventional tricks without emphasizing their implementation-dependent nature do a disservice to their readers. Programmers who use the tricks when not absolutely necessary design obsolescence into their code.

Work Remaining

The casual analyses done here are based on the unrealistic assumption that T (the target line) and P (the current line) are independent, random variables. Better definitions of T and P would allow better analyses and perhaps lead to the discovery of better methods. It should be possible to trace the dynamic behaviour of a sample of real BASIC programs to arrive at a better definition of T and its relation to P. Such a study would make an excellent article for *DDJ*!

The speed of Method 8 depends critically on the hash function used. What is a good hash function for use with BASIC line numbers? What are the characteristics of line numbers as used in real programs, and can the hash function take advantage of them? **DDJ**

```
Method_2:

    global addresses A, M, P, Z
    local addresses S, Q

    if ( P<M ) then
        if ( T<NUM(P) ) then { backward in upper half }
            S := A; Q := P       { n/4 }
        else
            if ( T<NUM(M) ) then { forward in upper half }
                S := P; Q := M       { n/4 }
            else                 { forward to lower half }
                S := M; Q := Z       { n/2 }
            endif
        endif
    else
        if ( T<NUM(P) ) then { backward in lower half }
            if ( T<NUM(M) ) then { backward to upper half }
                S := A; Q := M       { n/2 }
            else                 { backward in lower half }
                S := M; Q := P       { n/4 }
            endif
        else                     { forward in lower half }
            S := P; Q := Z       { n/4 }
        endif
    endif

    while ( NUM(S)<>T ) and ( S<Q ) do SCAN(S)

    if ( NUM(S)<>T ) then ERROR
                     else P := S

end Method_2.
```

Figure 3

If you want to follow up one of these announcements, look in the text for a name in **bold-faced type**. That name is the contact point for the product; you'll find it in alphabetical order in the Contact Points box at the end of the column.

Adorable Drives

We're in love with a photograph – the picture of **Seagate Technology's** ST706 disk drive. This has to be the most adorable piece of hardware we've ever seen. It is a hard disk with a removable media cartridge, and it fits in the space of a 5-inch diskette drive. Until now, hard disks for personal systems have had a very serious disadvantage: you couldn't move the media around. That made it hard to back data up, and created all sorts of other operational difficulties. The ST 706 goes a long way toward solving all these problems; it appears to offer the convenience of interchangeable cartridges with the speed and capacity of a hard disk. The metal disk in the cartridge holds 5 megabytes (formatted). Can you imagine slipping a pair of these little jewels into your TRS-80 III? Your North Star? Your IBM PC?

Well, don't hold your breath. We tried hard to get more info, but Seagate isn't ready to be specific about the ST706. The device exists in prototype, and should be available in sample quantities near the end of 1982. Quantity production won't begin until 1983. They refused to speculate about the price of the drive. And they were mum on the most crucial question of all – the price of the cartridge. If that isn't cheap enough to be bought in quantity by individuals, the little beauty will have priced herself out of our reach. Pray, brethren. Better yet, write to Seagate. Tell them how much you want an ST706, and what you think a cartridge ought to cost – say, the same as the box of quality diskettes it replaces.

Spring Out to a Conference

It's to be an event-packed Spring. Readers in the Baltimore, MD area may see this in time to attend the "Hamboree and Computerfest" on March 28th; it will feature speakers, a "large" vendor area, and flea market spaces. Contact **GBH&C**.

Computer makers will be putting on their three-button armor and their businesslike faces in preparation for the 1982 Office Automation Conference, sponsored by **AFIPS**. It will be held in San Francisco's new Moscone Center April 5-7.

Then, May 14-16, East-coast Apple fans have the Boston Applefest to entertain them. It's run by **National Computer Shows**, who claim it will be "the largest Apple-specific computer show in the country."

The National Educational Computing Conference **(NECC-82)** will be held June 28-30 in Kansas City, MO. It's too late to submit a paper, but you might still be able to reserve exhibit space; try **Jon Rickman**.

Spring Off to the Far East

Could you pass for a "top electronics executive"? Would you like to check out the Japanese and other far-East competition at first hand? Then you should join the Spring Electronics Tour; it will visit the Tokyo Communications show, the Hong Kong Electronics show, and points in South Korea and Taiwan. There is an optional extension to the Canton Trade Fair and another to Beijing, both in the PRC. Call **Ellen Chang** and be ready to leave April 4th. Of course, if you go, you'll miss **Computer Swap America** in San Jose, CA that same day.

Sorting Cinema

Ronald Baecker and David Sherman of the Unviersity of Toronto's Dynamic Graphics Project have produced a 30-minute film on computer sort algorithms. The entire film is computer-generated animation, in color, with an original musical score. From the stills included in the brochure, they've done an outstanding job of visualizing the operation of the nine different algorithms they chose. It looks like a very desirable item for club, classroom, or library use. The film is available in 16mm or any of three videotape cartridge formats. No prices were given, but it can be purchased or rented through the **Distribution Office**.

Flock Together

We've had notification of two new User Groups. The **Autumn Revolution** is a romantically-named association of users of the IBM Personal Computer. It aims to do all the usual things – have a newsletter, build a software library – and also to have a technical hotline phone. Who'll man it? Dues are $30 a year. The release contained no phone number, nor any personal names of the organizers.

Cary Davids is forming a User Group for those who own the SD Systems Z80 Starter Kit. A newsletter, *ZSTART*, is the main group product. No dues are mentioned.

(Parenthetically, isn't "<product> User Group" a better phrase than "<product> Users Group," or "Users' Group," or "User's Group"? There's a great temptation to put an "s" on "User," and then it seems to need an apostrophe. The Autumn Revolution people write "Users Group," but that's redundant; a group has multiple members by definition. Cary Davids writes "Users' Group." While grammatical, it says that a number of Users *own* the Group. They don't; they *are* the group. "User's Group" implies that there is only one User – he has a gang, perhaps, but they don't Use. No, we think that a *Group* of *Users* is a *User Group*.)

Insider's Info

Sources of information for people with various professional interests in computing: **Custom Cassette** says that they both manufacture, and duplicate the contents of, cassette tapes. They claim that because they do both, they do it all better and maybe cheaper . . . **Market Data Retrival** prepare CIC's School Directories, which list each school, public or private, elementary or high, in the nation, state by state. Locations, enrollment, names and numbers of principals and district officers – everything you could need to market a product direct to educators. A typical one-state volume costs $32 . . . **Barbara Schiffman** edits *W.P. News*, a bimonthly newsletter which aims to inform professional writers using, or contemplating the use of word processors. It offers, she says, equipment reviews, user interviews, and idea exchanges . . . **Vogeler Publishing** produce *Micro . . . Publications in Review*, a monthly quick reference to articles in "70+ Micro, Mini, and Technology publications." They reprint tables of contents and cross-index by subject, computer, and operating system. A subscription is $56 for 12 months . . . **DP Directory** is similar, but it apparently aims to cover the whole data

processing field and doesn't mention index categories; $48 for 12 months . . . and **Elliam Associates** have updated their $20 database of CP/M articles to include 1981 entries.

OK, Richard . . .

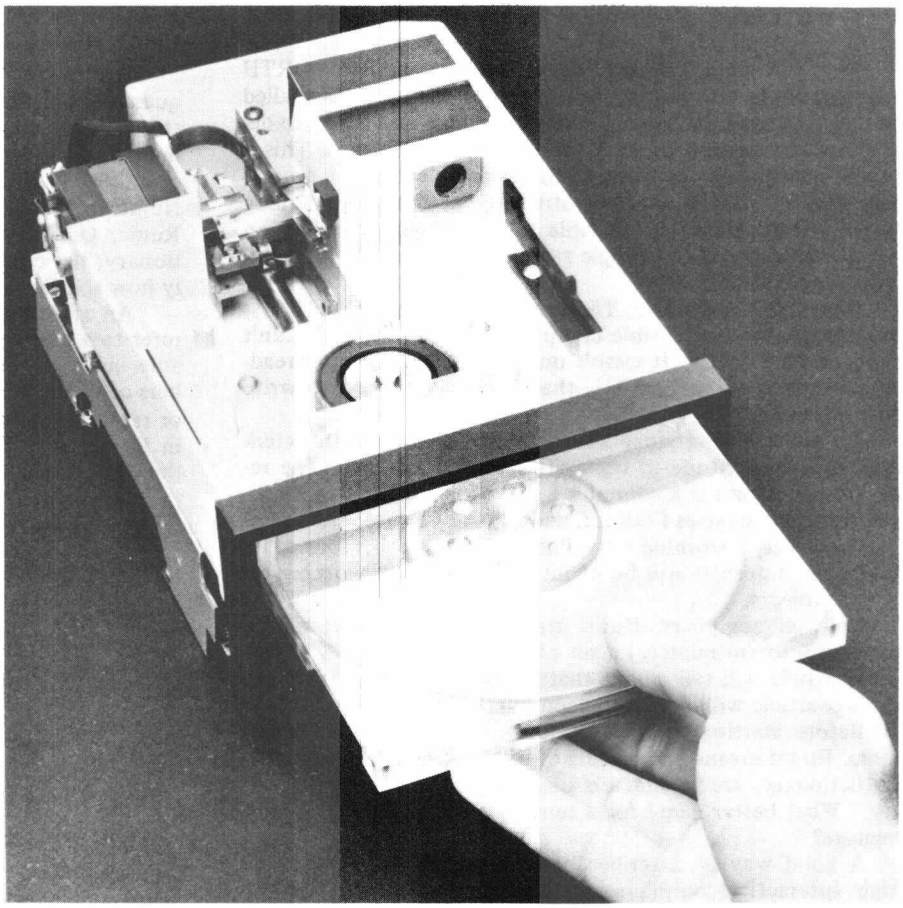
Richard Roth, vice president of InfoSoft, writes us a plaintive personal letter. "We've been subscribers of *Dr. Dobb's* since its inception . . . to date, not one of our press releases has been printed . . ." The releases he enclosed do look interesting. **InfoSoft**'s primary product, it appears, is a pair of CP/M-compatible operating systems, I/OS and MultI/OS. The latter is a multi-user system supporting up to 16 concurrent tasks; it can be run on an enhanced TRS-80 Model II, among other systems. They have networking software for linking clusters of systems, and a smart-terminal package (I/TERM) that can be driven by scripts prepared off-line, allowing minimal connect-time.

RAT-BAS?

An outfit called **TerraSoft** has produced FloBASIC, a structured preprocessor for BASIC code. As with the well-known RATFOR preprocessor for FORTRAN, FloBASIC allows the programmer to write with a "set of control constructs very much like those used in Pascal and C." The program translates those into ordinary BASIC suitable for "BASIC compilers and interpreters," which ones we don't know. The price for what could be a most useful product is $75.

Join a Mainframe Plot

Among mainframers, CalComp is a big name among plotters, and CalComp's Basic Subroutine Package (BSP) is a *de facto* standard for plotting software. Enter **Tesseract Associates**, who have created QCAL. That's a set of software tools that emulate the BSP on a CP/M system. With QCAL and Microsoft's FORTRAN you can "employ the standard calls to the CalComp BSP, but utilize many other graphic output devices." The $195 package can be ordered with a device driver for Houston Instrument's HIPLOT or for an NEC Spinwriter, and it comes with instructions for writing your own device driver. ▶▶J



The Seagate ST706, a cartridge disk drive that excites lust in the hearts of hobbyists.

Contact Points

AFIPS, 1815 N. Lynn St., Arlington, VA 22209; (703) 558-3612.

Autumn Revolution, POB 55329, Tulsa, OK 74155.

Ellen Chang, Commerce Tours International, 870 Market Street, Suite 742, San Francisco, CA 94103; (415) 433-3072.

Computer Swap America, POB 52, Palo Alto, CA 94302; (415) 494-6862.

Custom Cassette Cartridge Company, 6228 W. Oakton Street, Morton Grove, IL 60053; (312) 478-2620.

Cary Davids, 6000 Puffer Rd., Downers Grove, IL 60516.

Distribution Office, University of Toronto Media Centre, 121 St. George Street, Toronto, Ontario, Canada M5S 1A1; (416) 978-6565.

D. P. Directory, POB 562, Bloomfield, CT 06002.

Elliam Associates, 24000 Bessemer Street, Woodland Hills, CA 91367.

GBH&C, POB 95, Timonium, MD 21093; (301) 561-1282.

InfoSoft, 25 Sylvan Road South, Westport, CT 06880; (203) 226-8937.

Market Data Retrieval, POB 510, Westport, CT 06881; (203) 226-8941.

NECC-82, E. M. Staman, General Chairman, University of Missouri-Columbia, 305 Jesse Hall, Columbia, MO 65211.

Jon Rickman, NECC Exhibits Co-ordinator, Northwest Missouri State University, Computing Center, Maryville, MO 64468.

Barbara Schiffman, 1765 N. Highland #306, Hollywood, CA 90028; (213) 463-4171.

Seagate Technology, 360 El Pueblo Rd., Scotts Valley, CA 95066; (408) 438-6550.

TerraSoft, Dept. M, 25 Bryan Road, Rowayton, CT 06853; (203) 852-0374.

Tesseract Associates, Stinson Lake Road, Rumney, NH 03266; (603) 786-9561.

Vogeler Publishing, Inc., 455 Crossen Ave., Elk Grove Village, IL 60007; (312) 228-0951.

Runic: An Interactive, Extensible Compiler

After I heard about them, I wanted an extensible compiler, too.

A what, you ask? An extensible compiler. FORTH is an example of one: it has a set of instructions, called "words," that control a virtual stack machine and a way to define new instructions based on those that already exist. This is a bald description, of course; the salient feature of a compiler like FORTH is that the structure of the programming language it translates is accessible to the programmer. He can invent new data types or logic structures as needed for a particular application.

Boy, did I want one. The problem was that when I first found out about extensible compilers, back in 1977, I didn't know about FORTH. It wasn't until this last year, after reading about and using FORTH, that I learned enough to write these articles.

To make a long story short, my fascination with extensible compilers led me to try writing one of my own. The result of my efforts is a compiler I call Runic. What I've developed isn't the same as FORTH, and I make no apologies for it. It is, however, a working compiler, and I hope this tour of its design and internals will be of interest to readers working on similar projects.

Like all compilers, Runic translates a high-level source language into (ultimately) a set of machine actions. This particular article will talk about that source language's structure. My next article will discuss the compiler's construction.

Before starting, though, a quick word about the name, Runic. Runic means "consisting of runes" which, according to my dictionary, are "characters of a magical or mysterious nature." What better name for a terse, RPN-based programming language?

A good way to describe Runic is through its virtual machine. Interactive compilers create a machine in software that the programmer controls with his source statements. This software machine is easier to use than the actual hardware, since it usually works with higher-level objects like strings and variables rather than bytes and addresses. Runic is no exception to this practice. The "Runic machine" consists of:

- A data structure containing the names and definitions of all the words (instructions) that the Runic machine knows at any point in time. This is called the "dictionary."
- A last-in, first-out data stack for keeping intermediate data during execution. This is simply called the "stack."
- A global condition code tested by Runic words to allow conditional execution. This is called the "condition flag." The condition flag, stack, and dictionary together are the Runic machine's store. Lastly, there's:
- The current statement the compiler is processing.

Now let's clear up the fuzzy notions of "statements" and "words." A Runic statement is a sequence of words, separated by blanks and terminated by a comment or newline. Comments begin with an open parenthesis, and "newline" means the combination carriage-return and linefeed (CRLF).

By "word," we mean either an integer constant (a sequence of digits, possibly beginning with a sign), a quoted string (a sequence of characters enclosed in double quotation marks), or what's called an "ordinary word" (a sequence of characters that isn't an integer constant or quoted string). Numbers (integers) and strings are the basic data types in Runic. Ordinary words are the names of entries in the dictionary; the compiler has to look them up to determine exactly how to interpret them.

An ordinary word can have one of several meanings. It can refer to a numeric or string data object and thus act like a variable in other programming languages. Words defined as numbers or strings must first be declared; they can then be assigned or recalled as needed. Below is an example of how they're used in Runic. The character ">" is Runic's prompt, the text after the open parenthesis is a comment.

```
> "total" number ( This declares an integer variable
> "myname" string ( This declares a string
10 ↑total "Runic" ↑myname ( Assign them values .
> total ? myname ? (Recall and print their values
10
Runic
>
```

Values are assigned to variables by using a variable's name with an assignment operator ("↑") placed directly in front of it with no blank. This takes whatever is topmost on the stack currently and stores it as the variable's definition. To use the variable's contents in a statement, we just use the variable's name as an ordinary word. When the compiler scans "total" and "myname" in the last statement, it will automatically push their contents back on the stack. Nowhere in Runic does a user need to explicitly know a variable's address.

The "?" is an example of another type of ordinary word. If we enter

```
> "?" show
```

we get back

```
    ( Size is 7 bytes )
    Say
    ;
>
```

"?" is a user definition: a word defined as a list of other Runic words. User definitions are extensions to the language created by the programmer. They are entered with the ":" word, shown below. The words "say" and "." are examples of still another kind of ordinary word, one that is written in the target machine's language. These types of words are called ker-

by Marty Franz

Copyright © 1981 by Marty Franz.
Marty Franz, 525 W. Walnut St., Kalamazoo, MI 49007.

nel functions, and they are the bedrock of functions that the Runic machine supports. "Say" is used to display whatever is on top of the stack, regardless of type, and the "." word prints a newline.

A user definition can contain any sequence of Runic words: numeric constants, quoted strings, other user definitions, or kernel functions. Comments, if they are entered as part of the definition, are stripped off and now saved – source can be annotated in disk files if need be. If a word isn't in the dictionary at the time it is referenced, Runic creates a dummy entry for it and lets the current definition continue. This allows a set of user definitions to be written top-down, with a "main" definition calling sub-definitions. This forward-reference ability doesn't apply to variable assignments, though, since Runic has no typing information about variables that haven't been declared yet. When this happens, an error message is printed and the definition is cancelled.

Here's how a user definition is entered, using the statement we used earlier to demonstrate variables:

```
> "printvars" :
   total ?
   myname ?
   ;
> printvars ( Now test the new word
10
Runic
>
```

A definition is ended by entering a semicolon (";"). When the definition is ended it is added to the dictionary and becomes available for use in statements.

With the Runic machine now essentially described, here's a summary of the rules of scanning statements:

1. Get the first word in the statement.
2. If the word begins with an assignment operator, the topmost data item on the stack is popped and stored as the word's definition (assuming types match). Go to step 7.
3. If the word is a numeric constant or quoted string, push it on the stack. Go to step 7.
4. If the word is a variable, push its contents on the stack. Go to step 7.
5. If the word is a kernel function, execute it. Go to step 7.
6. If the word is a user definition, then the words that make up its definition become the new current statement. Go to step 1.
7. Get the next word in the statement until all the words have been processed. Return to the previous statement, or to the console if there is none.

Normally, processing goes sequentially from left to right through the statement. This quickly leads to useless programs unless there is the ability to test the virtual machine's state and alter control flow, so Runic has kernel words for doing just that.

Runic has two basic control structures, conditional and iterative. They resemble control structures of Pascal or C rather than FORTH: namely, they are not RPN-based. The conditional structure is the traditional if-then-else:

```
If <condition> then
   <words1>
Else
   <words2>
Endif
```

where <condition> is a sequence of words that sets the condition flag to a "true" or "false" value. Scanning continues with the sequence <words1> if the condition flag is true, or jumps to <words2> if not. The "else" part of this structure is optional; when omitted, if the condition flag is false, scanning resumes with the word immediately after the Endif.

The iterative structure is the do-end. The terminating condition can be specified by placing a "while" or "until" word before the <condition> words. The "while" structure looks like this:

```
While <condition> do
   <words>
End
```

As long as the <condition> words set the condition flag "true," <words> will be scanned. As with "while" loops in most programming languages, evaluation occurs before the loop body is executed. The "until" flavor is similar to "until" loops in other languages, too: conditional evaluation occurs after the loop body has been executed. Scanning of <words> in an "until" loop will occur as long as <condition> sets the condition flag "false."

Here's a sample definition that uses these two types of control structures. It prints a list of numbers from 1 to 50 in tabular form with a newline every 5 numbers. First the main definition is entered:

```
> "table" :
   1
   while dup 50 <= do
      dup say
      tab
-  Tab word created
      if dup 5 mod 0 = then ( "mod" is remainder )
         . ( Print newline )
      endif
      1 + ( Add 1 to top of Stack )
   end
   drop
```

```
;
```

and then the definition for "tab":

```
> "tab" :
   9 >char say ( ">char" makes ASCII code a string
   ;
```

The "tab" word is worth an extra mention. It's an example of forward-referencing in Runic. The compiler, confronted with a reference in "table" to a word that didn't exist, went ahead and created a dummy reference to "tab." This is what the message

– Tab word created

indicates. Executing "table" at that point would have resulted in an error message when the word "tab" was scanned. After "table" was completed, "tab" was filled in and that resolved the undefined reference. Running "table," we get

```
> table
 1    2    3    4    5
 6    7    8    9   10
11   12   13   14   15
16   17   18   19   20
21   22   23   24   25
26   27   28   29   30
31   32   33   34   35
36   37   38   39   40
41   42   43   44   45
46   47   48   49   50
>
```

Going back to the definitions again, notice that only the stack was used as a data store: neither "table" nor "tab" reference any variables. This is a feature of RPN languages that is often abused, so in the example shown the stack's use is (more or less) direct.

The Runic virtual machine, the language's control structures, and some kernel functions have been presented. This was the source language for which I wanted to write my compiler. Unfortunately, when I began work, many of my ideas were incomplete or ill-considered. For example, I had no idea how I was going to allow forward-referencing dictionary entries. All I knew was that I wanted the ability to enter and test definitions top-down. As it turns out, allowing pending or incomplete definitions (like "tab" in the last example) critically affected the dictionary's structure. But I'm getting ahead of myself.

I'll end here for now by saying that, incomplete ideas or not, I went forth (!) to write Runic. I'll begin telling how in my next article. **DDJ**

Put the H-8 Front Panel to Work

In this day of computers-in-a-box, one of the most distinctive features of Heath's H8 remains the front panel, with its keypad and nine seven-segment, light-emitting diodes (LED's). The LED's are especially eye-catching when the H8 is equipped with a disk system, as they continually reflect the sweep of the stack pointer or the program counter (or any other register pair or memory location you might want to keep an eye on). However, with just a little effort, these LED's can be used to print out messages, display information, and in general spice up your computer operations, all while running BASIC, C, Pascal, FORTRAN, tiny BASIC, FORTH, or any other language that allows you to send a byte out to an address in the computer's memory.

The H8's front panel displays information in three groups, with the numerical data usually shown in split octal notation, and the register name by means of two letters (SP, HL, DE and so on). The register identification is replaced with the contents of memory whenever the front panel is set to display a memory location in the two leftmost groups. It should be noted that it's simple to modify the disk operating system to allow display of data and addresses in hexadecimal, for those of you who prefer that notation. If you make a minor modification to the Heath assembler, you can do all your work in hex instead of split octal. I mention that only because many people have made the comment to me that it was regrettable that Heath chose to use only split octal. Not true; you do have a choice.

For the purposes of this article, we will restrict our choice of language to BASIC, primarily to make it easier to illustrate the techniques used, but also because virtually every H8 owner has at least one version of BASIC; whether he or she uses it for much of anything is another matter. We will also make use of a feature peculiar to Heath's Benton Harbor BASIC, the CNTRL-O command used to jump to a special subroutine. This isn't really very important, but it makes the demonstration more effective. Ignore it if you're using Microsoft BASIC or some other language.

Now for a bit of background. Heath's Benton Harbor BASIC, like most other extended BASIC's, either disk or tape, supports the POKE command that allows the user to send a byte to any of the computer's memory locations. A little research through the Heath manuals reveals the fact that the LED's contents are contained in addresses 8203 through 8211 (decimal). This makes it easy to control the display. The next useful bit of information is that the LED's will respond to any decimal digit POKE'd out into them. There are 255 patterns that each LED will produce; the chart in Figure 1 shows what each one is.

Figure 1 requires a brief explanation. It's arranged as a matrix, such that the decimal digits across the top and down the left side give the patterns. To determine what numbers produce what patterns, it's only necessary to read over to the intersection point. For instance, decimal 129 gives a hollow rectangle that could be a zero or the letter "O". To see that, read across the top to the least significant digit (9, in our example) then down to the other digits (12, in this case). Simple, when you get used to it.

Now to the listings. Listing 1 is a demonstration routine that generates all possible patterns, and displays on the terminal the decimal number which produced that pattern. In this case, all six left-most LED's produce the same pattern. This need not be the way you implement the program; it was just easier to write that way. The routine does its work in two ways. First, it will produce all the patterns automatically, with no operator intervention. It makes use of the CNTRL-O feature of B-H BASIC to interrupt the generation and hold the pattern for examination. When you tire of that approach, however, you may use the other mode of operation and enter a decimal number to check what the pattern looks like. This routine is probably best used to become familiar with which patterns can be produced, and what they look like on the front panel. All look good, but obviously some letters come across better than others.

Listing 2 is a simple application of what we've seen so far. It is used to send a travelling display across the H8 front panel, from right to left, theater-marquee style. The phrases are simple, but easily read, and can be altered as to timing and sequence so that they come closer together or farther apart. By changing the data statements at the end of the program, any desired message can be sent. This is a technique that can be readily incorporated into any of your BASIC programs, especially since the only "exotic" feature of BASIC you need is the POKE statement, or the equivalent command in whatever language you choose.

If you have an H8, you have a front panel. Why not make use of it? Few other systems give you the degree of freedom and flexibility available with the H8. For me the front panel is the most valuable single hardware feature of the entire system.

Technical Addendum

For those who are interested in more fully utilizing the H8 front panel, here are some other memory locations you can POKE. The first, 8199, controls the memory display. To use it, POKE 8199,1. To display a register, POKE 8199,2. Then POKE 8197,N where the value of N is selected from the following:

N	Register
0	SP
2	AF
4	BC
6	DE
8	HL
10	PC

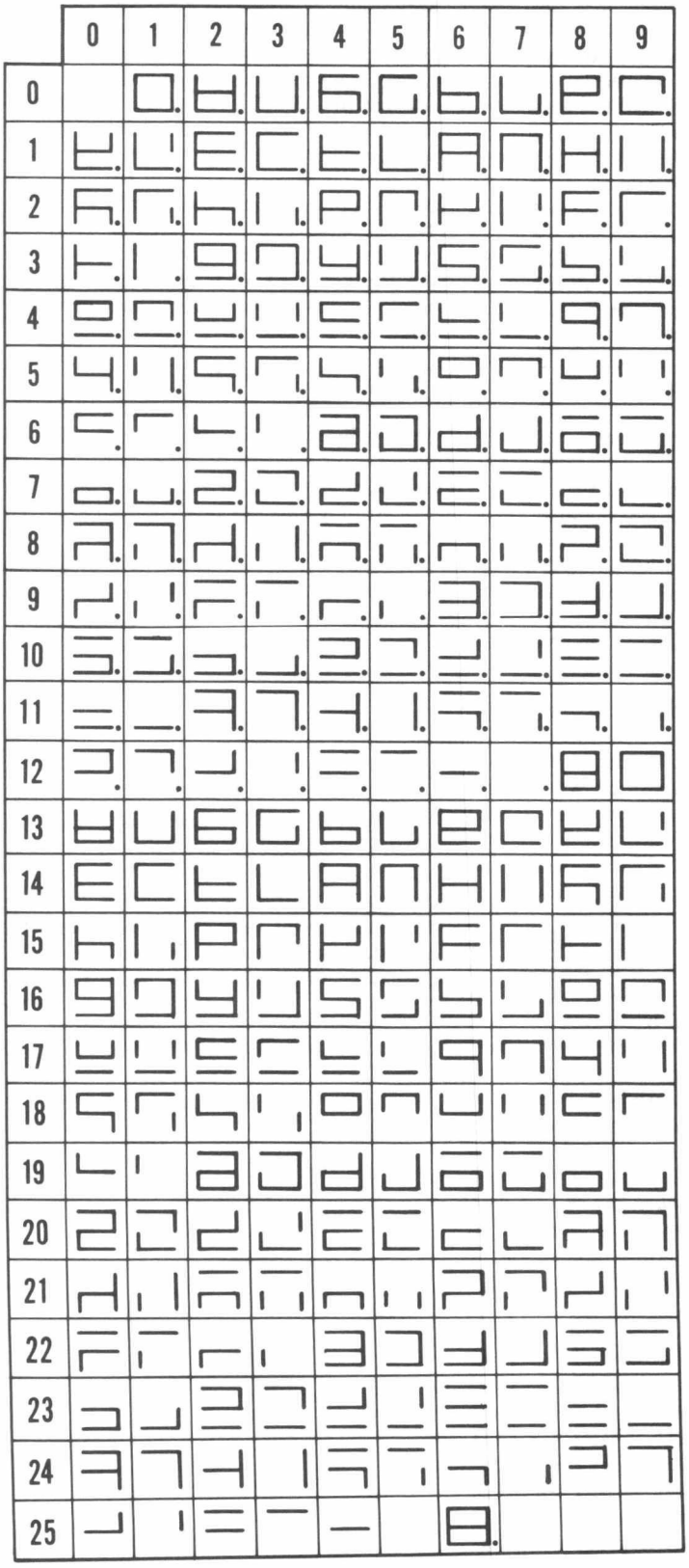
By selecting the appropriate values of N, you may call up any information you want on the front panel. This can be especially useful in debugging and in interfacing work, and in the incorporating of machine-language subroutines into programs written in BASIC and other languages. ▶▶J

(Listing begins on page 34)

by D.C. Shoemaker

D. C. Shoemaker, 2000A Foxridge, Blacksburg, VA 24060.

Figure 1. Possible LED pattern combinations



Improvement Upon a Division Program by Leventhal

by W.D. Maurer

W. D. Maurer, George Washington University, S.E.A.S., Washington, DC 20052.

The purpose of this paper is to give an improvement in timing, without affecting memory requirements and by changing only two instructions, in a 16-bit-by-8-bit division program for the 6502, due to Leventhal (Ref. 1, pp. 8-15).

The original algorithm is reproduced in Figure 1. It assumes that the 16-bit number in hexadecimal locations 40 and 41 (with bytes reversed, as usual) is being divided by the 8-bit number in location 42 to produce a quotient in location 43 and a remainder in location 44. All quantities are unsigned, but the leftmost bit of location 41 and the leftmost bit of location 42 are both assumed to be zero; this would be the case if the division were originally a signed division and the absolute values of the signed quantities had been taken. Overflow and division by zero are not checked.

The improvement is concerned with eliminating the INC instruction. Notice that INC adds one to the quotient, thus setting the current bit of the quotient to 1. If the INC is not done, the current bit of the quotient is zero.

Notice, however, that if INC is executed, SBC has just been executed, and this subtraction has produced a nonnegative result. Hence the carry bit will be equal to 1. On the other hand, if INC is skipped, by branching on carry clear to CHCNT, then the carry bit will be equal to zero.

Hence, in either case, the carry bit will be equal to the current bit of the quotient. This will be true at CHCNT, and therefore also at DIVID when we go back to the start of the loop, so that the bit can be rotated into the quotient by changing the ASL $43 to ROL $43 at this point.

The last time through the loop, the final bit of the quotient will still be in the carry bit, so that an extra ROL $43 is necessary immediately following the BNE. The two instructions changed are therefore:

(1) Change ASL $43 to ROL $43.
(2) Eliminate INC $43 and put another ROL $43 between BNE DIVID and STA $44. (All of the four instructions above have the same length, so that the memory requirements of the program are not affected.)

Note that the *first* time through the loop, the ROL at DIVID inserts a nonsense bit into hexadecimal location 43. This nonsense bit is then rotated all the way through this location and becomes the final value of the carry bit when the

Figure 1

```
        LDX   #8        ;NUMBER OF BITS IN DIVISOR = 8
        LDA   $40       ;START WITH LSB'S OF DIVIDEND
        STA   $43
        LDA   $41       ;GET MSB'S OF DIVIDEND
DIVID   ASL   $43       ;SHIFT DIVIDEND, QUOTIENT LEFT 1 BIT
        ROL   A
        CMP   $42       ;CAN DIVISOR BE SUBTRACTED?
        BCC   CHCNT     ;NO, GO TO NEXT STEP
        SBC   $42       ;YES, SUBTRACT DIVISOR (CARRY = 1)
        INC   $43       ;AND INCREMENT QUOTIENT BY 1
CHCNT   DEX             ;LOOP UNTIL ALL 8 BITS HANDLED
        BNE   DIVID
        STA   $44       ;STORE REMAINDER
        BRK
```

final ROL is done.

This fact suggests a further improvement in the program. We have noted that overflow and division by zero are not checked in this program and therefore must be checked externally; this takes at least two instructions. Suppose, however, that we were to check this in the given division routine itself. These two instructions would then be moved into the routine from outside it, thus causing again, no change in space requirements. The instructions might be CMP $42 and BCS ERROR inserted just before DIVID. (This works because the contents of location 41 are in the A register at this point. If the contents of location 42 are less than or equal to this, the carry will be set, and we have either the division-by-zero case, where location 42 contains zero, or the overflow case.)

Now notice that, in this case, the nonsense bit, as mentioned above, will always be zero, because when the first ROL is done the carry will be clear (otherwise we branched on carry set). Therefore the final value of the carry bit will also be zero, after the nonsense bit has passed through location 43. On the other hand, in either of the two abnormal cases, the carry will be set.

The further improvement is now as follows. Instead of branching to ERROR on carry set, we simply branch to the end of the program. We can now make the program into a subroutine by adding an RTS, and since this leaves the carry flag unaffected, it can serve, in the *calling* program, as an indication of error. Thus, for example, if the name of the subroutine is DIV, then JSR DIV followed by BCS ERROR will serve, in the calling program, to go to ERROR in either of the error cases and continue in the normal case. The program with all these improvements is illustrated in Figure 2.

If only the initial two changes are made, the timing improvement will be $5(n-1)$ cycles, where n is the number of one-bits in the quotient. The first change makes no difference; the new ROL takes 5 cycles, outside the loop, while the eliminated INC takes 5 cycles for each one-bit in the quotient.

■■J

Reference

[1] Leventhal, L. A., *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, California.

Figure 2

```
DIV     LDX   #8        ;NUMBER OF BITS IN DIVISOR = 8
        LDA   $40       ;START WITH LSB'S OF DIVIDEND
        STA   $43
        LDA   $41       ;GET MSB'S OF DIVIDEND
        CMP   $42       ;SHOULD BE LESS THAN DIVISOR
        BCS   DONE      ;IF NOT, ERROR EXIT (CARRY = 1)
DIVID   ROL   $43       ;SHIFT DIVIDEND, QUOTIENT LEFT 1 BIT
        ROL   A         ;(AND NEW ANSWER BIT -- SEE DEX BELOW)
        CMP   $42       ;CAN DIVISOR BE SUBTRACTED?
        BCC   CHCNT     ;NO, GO TO NEXT STEP (CARRY = 0)
        SBC   $42       ;YES, SUBTRACT DIVISOR (CARRY = 1)
CHCNT   DEX             ;NOTE CARRY = NEW ANSWER BIT
        BNE   DIVID     ;LOOP UNTIL ALL 8 BITS HANDLED
        ROL   $43       ;SHIFT IN THE LAST ANSWER BIT
        STA   $44       ;STORE REMAINDER (CARRY = 0 HERE)
DONE    RTS             ;QUIT (CARRY 0, NORMAL, CARRY 1, ERROR)
```

H-8 Front Panel (Listing continued, text begins on page 17) *(Continued from page 35)*

```
390 POKE A+2,255:POKE A+1,L

400 PAUSE 20

410 POKE A+1,255:POKE A,L

420 PAUSE 20

430 POKE A+1,255:PAUSE 20:POKE A,L

440 PAUSE 20

450 POKE A,255

460 NEXT K

470 REM Number of characters to print

480 DATA 27

490 REM Data for characters

500 DATA 146,254,128,164,144,162,164,146,140,143,143,129,255,159,200,224,178

510 DATA 164,132,241,128,176,129,255

520 DATA 134,162,136,255

530 END
```

End Listing II

Letters *(Continued from page 5)*

historic accomplishments presented by Raphael and the Nova tape, perhaps a letter to David Touretsky, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, PA 15213 will convince this AI researcher/instructor to release his rough draft to you for some reasonable price.

Hofstadter (*Godel, Escher, Bach* of course) gives a nice example of recursion in language generation beginning on page 130. The charts here mirror those commonly used by Pascal hackers, so P-coders, welcome to the AI programmers' circle! The discussion on chess players is also quite clear, p. 150.

An AI/Robotics group has started up here again; interested parties willing to exchange information can write to me.

Good luck!

Abby Gelles
185 W. Houston St.
New York, NY 10014

Modem Routine on RCPM

DDJ:

I have rewritten the receive routines I used ("Terminal Programs for North Star," *DDJ* #61) in MODEM.PAS so they no longer rely on X-ON/X-OFF to control the transmission of data from the PRIME. The changes allow me to receive data at the same rate I can transmit and with more reliability. I have also modified MODEM.PAS to work with a PMMI-103 modem and added auto-dial, retry, and baud rate change. I placed a copy of this version on the Hyde Park RCPM/RBBS in Chicago (312-955-4495).

Sincerely,
Richard Blessing
Rt. 4, Box 5
Geneseo, IL 61254

Editor Notes

Dear Sirs:

Enclosed find my check for $39.00, please renew my subscription. I had really felt that the magazine had somehow lost that fantastic ability to have at least one or two articles in each issue that were exactly what I had been looking for, or had not even thought of. The January issue arrived this morning and you have done it again. Not two days ago I was grumbling about needing to get some sort of screen editor written, and now it looks like Ream's editor will do most of what I need.

Some of the points brought up in the article on the 68000 are very relevant, in particular the statement about the lack of software published for the sixteen-bit machines. I am working on a small Z8000 design and hope to get it running soon. This all depends on failing to improve the design to a point that it never gets built. If I can restrain myself, I hope to submit a monitor-debugger to you. This should be followed by a simple, resident assembler, and patches to Cain's C compiler, with no register allocation optimizing, yet. I must warn you that my track record is only fair, and I make no guarantees.

Keep up the good work.

Sincerely,
Don Taylor
627 S.W. Third #3
Corvallis, OR 97333

DDJ

A Pattern Classification Program*

*Based on Fisher's Methods

by K.F. Wegner and D.E. Raeside

Work supported in part by funds provided by Richard and Addie Fleischaker.
K. F. Wegner and D. E. Raeside, Department of Radiological Sciences, University of Oklahoma Health Sciences Center, Oklahoma City, Oklahoma.

Pattern classification computer programs, such as the one presented in this paper, have applications in industry, science, medicine, and business. Their use in artificial intelligence systems is rapidly increasing.

Pattern classification involves the computer manipulation of measurable features of objects from various classes for the purpose of classifying the objects without human intervention. This paper describes a two-class pattern classification computer program written in Data General Extended BASIC which is suitable for use on small computer systems (the program contains Hazeltine 1420 CRT terminal screen control commands). The program assumes the availability of a sufficient number of objects of known classification (training samples) to permit samples of unknown classification to be classified automatically. The program uses Fisher's method (see pages 114-121 of Reference 1), a method of transforming complicated pattern classification problems into simple ones. As an example of the simplification afforded by Fisher's method, imagine a classification problem in which the two features F_1 and F_2 are used to classify samples. Each pair of measurements may be represented as a point in a two-dimensional plot of F_1 versus F_2 (see Figure 1). Fisher's method can be viewed as the projection of these points onto a line. When the direction upon which the points are projected is properly chosen, good class separation is obtained (as in Figure 2). Figure 3 shows the destruction of class separation produced by projecting the points onto some other direction.

The training data base selected to illustrate the authors' program consists of 100 samples, 50 from each of two classes. Each datum consists of measurements of the sepal and petal lengths and widths of an iris plant (see Figure 4) belonging to the species iris setosa or iris versicolor (the data are taken from Fisher's original paper, Reference 2). The projected point for each training sample is shown in Figure 5. An unclassified sample is shown in Figure 6 as a letter "A". It can be classified by simply noting which training cluster it is closest to. This particular unknown sample should be assigned to class 2 (iris versicolor). DDJ

References

[1] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* Wiley-Interscience, New York, 1973.

[2] R. A. Fisher. "The use of multiple measurements in taxonomic problems." *Annals of Eugenics*, 7 (1936) 179.

(Listing begins on page 37)

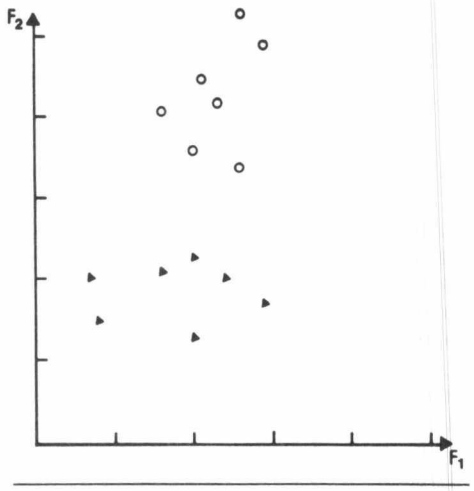


Figure 1.

A simple two-class pattern classification problem. The open circles represent data from Class I and the solid triangles represent data from Class II. The coordinates F_1 and F_2 are measurable features common to Class I and Class II objects.

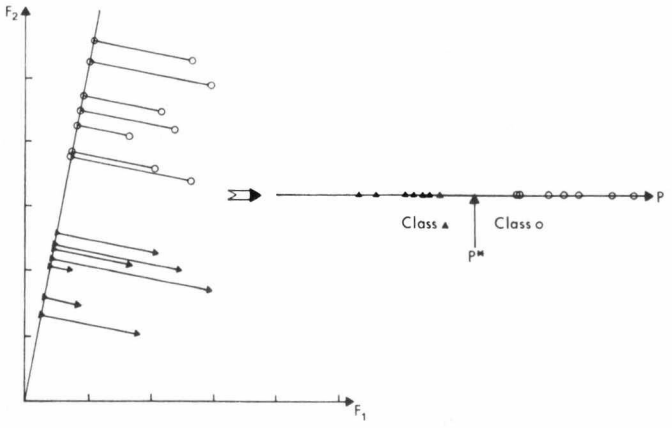


Figure 2.

An illustration of the simplification which Fisher's method provides for the pattern classification problem of Figure 1. Good class separation is obtained with a good choice of the direction upon which the data are projected (P* can be used as a boundary separating the two classes).

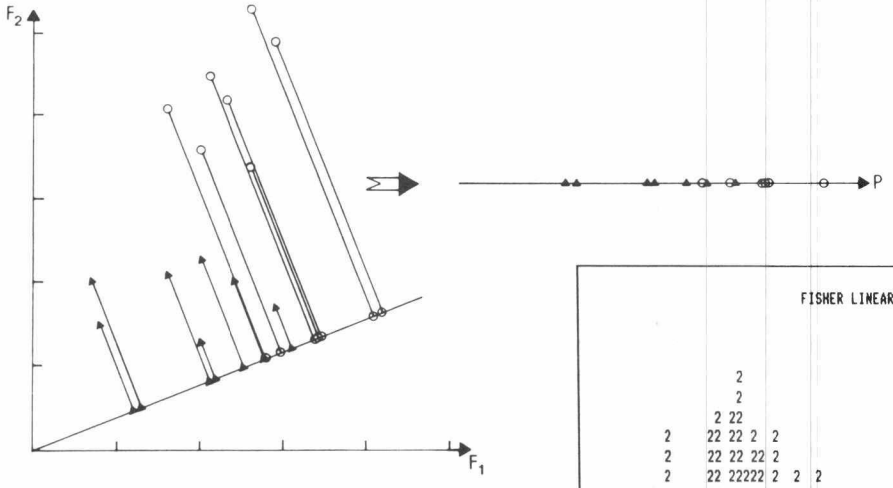


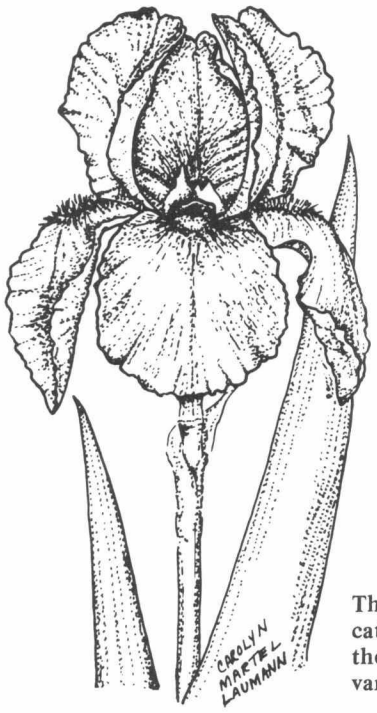
Figure 3.

The same pattern classification problem of Figure 1. Class overlap occurs with a bad choice of the direction upon which the data are projected.

```
                    FISHER LINEAR DISCRIMINANT PLOT

                                                          1
                                                          1
                      2                                1 11
                      2                                1 11
                    2 22                               1 11  1
    2     22 22 2  2                                   1 111 1
    2     22 22 22 2                                  11 11111
    2     22 22222 2  2  2                           111 11111 1   1
  2 2   2222222222222222                       1 11  111111111 11 11
    |****|****|****|****|****|****|****|****|****|****|****|****|****|
    |                         |                                   |
 -0.8835                   -0.1414                             0.6008

 SCALE FACTOR: EACH SPACE=    0.0212
 DO YOU WISH TO PLOT AN UNCLASSIFIED POINT?  Y
```

Figure 5.

The output of the authors' program for the training data.




```
                    FISHER LINEAR DISCRIMINANT PLOT

                                                          1
                                                          1
                      2                                1 11
                      2                                1 11
                    2 22                               1 11  1
    2     22 22 2  2                                   1 111 1
    2     22 22 22 2                                  11 11111
    2     22 22222 2  2  2                           111 11111 1   1
  2 2   22222222222222222  A                   1 11  111111111 11 11
    |****|****|****|****|****|****|****|****|****|****|****|****|****|
    |                         |                                   |
 -0.8835                   -0.1414                             0.6008

 SCALE FACTOR: EACH SPACE=    0.0212        UNCLASSIFIED POINT A=    -0.339504
 DO YOU WISH TO PLOT ANOTHER UNCLASSIFIED POINT? N
```

Figure 6.

Classification of an unknown iris sample using the authors' program.

Figure 4.

The object used by Fisher (Reference 2) to illustrate his method for pattern classification: an iris plant. The drooping parts of the iris are the sepals, the erect parts are the petals. The four features of the iris which are useful for classification of the various kinds of iris are the sepal and petal lengths and widths.

Pidgin for the 8080 Under CP/M

There are many ways to approach a problem and even more ways to solve it. This is by no means the only possible implementation and both Bill Gale and I invite others out there in the reader audience to make an idea better. How about an 8086 version?

– Andrew L. Bender

by Andrew L. Bender

Andrew L. Bender, M.D., Neurological Services, Inc. 336 Center Ave., Westwood, NJ 07675.

Copyright © 1981 by Andrew L. Bender, M.D. Permission is granted for individual, personal use in a non-commercial manner. All commercial rights reserved.

The recent publication of "Pidgin, A Systems Programming Language" in *Dr. Dobb's Journal* (*DDJ* #57) was an invitation to those interested in programming languages to prepare a specific implementation for a specific processor. After an initial look at things, I was tempted to implement my version of Pidgin by writing the macro processor, Tincmp, in assembly code using a macro assembler. This turned out to be more than a trivial task despite the excellent software tools which can be used under CP/M. Tincmp is a large program and there are many non-trivial bookkeeping tasks associated with the translation of this size program into assembly code. Not only are these tasks boring, but also checkout of such a large program is fatiguing and slow.

I initially investigated higher level languages, as were suggested in the *Dr. Dobb's* article. In both languages available to me, BASIC and FORTRAN-80, Tincmp would have been too cumbersome and difficult to get running. I had a copy of the new Digital Research PL/I-80 compiler and I knew that it would be easy to program the Tincmp compiler in PL/I-80 but I felt that my lack of experience with this compiler would make for slow going, and the average hobbyist would probably not have this compiler available. I found in Volume 51 of the CP/MUG library a program which was a relative of Tincmp. This program, STAGE2, was implemented by Dick Curtiss step-by-step from the text *Implementing Software for Non-Numeric Applications* by William Waite. If Tincmp was written in Pidgin and Pidgin could be compiled by a macro processor then, I reasoned, STAGE2 could compile Tincmp written in Pidgin if it were provided with a suitable set of STAGE2 macros designed to translate Pidgin. I would let STAGE2 do the translation while I learned about STAGE2.

A decision had to be made at this point as to how Tincmp was to implement Pidgin. It was easiest to take the assembly language approach. First of all, the 8080/Z80 processors do not lend themselves to the easy production and execution of relocatable programs. The 6502, like the 6800 or 6809 can easily run in an environment of relocatable programs because most of the instructions of these microprocessor series see memory as a relative set of locations rather than the more physical organization of the 8080 or Z80, whose organization depends on the actual operand address as a part of many instructions. Where the 6502 might approach an address as 5

Comments on 8080 Pidgin by William A. Gale

This is the first implementation of Pidgin for a non-6502 chip that I am aware of, and I am delighted to see it. OSI users may be interested to know that a version has been brought up on an OSI Challenger with tape storage by Walter Anderson.

Bender's decision to use Stage 2 rather than Basic is probably a good one for efficiency in the initial step. My Basic version (of a precursor to Tincmp) would have taken about one-and-a-half hours to compile Tincmp, while Anderson reported 10 hours for his Basic version of Tincmp! That Tincmp can compile itself in 3 minutes shows the efficiency of compiled code.

The advantage of being able compile Pidgin for a relocating linking loader are numerous. My own purpose in designing Pidgin included being able to write such a program, however. I have now written such a relocating, linking loader in Pidgin, called Maclod.

The 6502 relative branch instructions are not actually useful for Pidgin as compiled by Tincmp. The reason is that Tincmp has no idea how far ahead a transfer may be going. As a result, all branches become five bytes, and include a jump to an absolute location in the form

```
BNE 3
JMP absolute
```

for example. The relative branch just jumps over the absolute jump. The worst of both worlds! The 8080 three-byte test branches would be more efficient for Tincmp. A smarter compiler, such as Meta4, can get the two-byte branch where it is possible. This is done in a compiler for a high-level language, Spartan, now written for Meta4. However, a smart compiler is not written for Pidgin. Anyone interested?

The use of parameter statements to help isolate machine dependencies may be a good idea. I could have done better in the Tincmp code, perhaps, by putting all such assignments as the first statements of the main program. I did mark lines I considered system dependent with strings of exclamations. However, neither of these techniques is a complete solution, because the length of buffers is system dependent and is set by constants in Pidgin. A related problem is keeping the internal setting of a variable holding the length of an array matched with its declared length.

Bender's discussion of the use of prefixes to make unique labels should be useful to anyone else implementing Pidgin. You will find similar prefixes buried in the 6502 Pidgin macros, but no discussion of their use. If you are relying on Tincmp for the assembly stages of the compilation, making labels with prefixes diagnostic of the kind of statement needed to generate a definition helps in debugging. When you see F105 written out as a mismatched line in the third pass, for instance, you know that an ENDIF is missing in the source code.

The use of a special $$=+000 macro is an example of a slight optimization possible in the Tincmp framework. A slighter optimization in the 6502 setting would be to make a special macro for

```
I$$=+00000
```

as

locations from here in a forward direction or 7 locations back from here in a backward direction (forming the effective address in hardware during instruction execution) the 8080/Z80 needs to have the actual address of each operand for its memory reference instructions and, with a few exceptions on the Z80, even for its jump instructions. In the second place, the 8080/Z80 have available operating systems and operating-system-dependent programs which demand the physical loading of programs at certain memory addresses. CP/M, as an example, demands that the user programs be loaded into a transient program area which in the standard version of CP/M begins at 256 (100H). Other versions of CP/M were used but they had a different transient program starting address or "bias." If Pidgin was to be useful on all different systems it would be necessary to provide some convenient means of changing the address of the start of the program other than changing the HIMEM and LOMEM values. As a matter of fact, these values shouldn't appear in a relocatable program at all.

Some statements were therefore not implemented for the 8080/Z80 series. The statements not implemented are discussed here:

1. LOMEM This command is not used by this version of Pidgin and could be used in an absolute assembly to assemble an ORG card.
2. HIMEM This command is also not used. In an absolute assembly it might be used to assemble an ORG card for a data region.
3. REGISTER This command does nothing.

The specific implementation of Pidgin for the 8080 is slightly different than the implementation described by Bill Gale. Pidgin for the 8080 compiles to assembly code. This code is written to the output file in a format suitable for assembly with either Microsoft's MACRO-80 assembler or RMAC, Digital Research's macro assembler. Both of these assemblers produce relocatable code which can be positioned anyplace in memory when the code is link edited. The task of preparing relocatable code suitable for direct input to a linkage editor accepting a bit stream format (such as Microsoft's L80 or Digital Research's LINK-80) with Tincmp was investigated but felt to be beyond the capabilities of Tincmp. However, suggestions are given below for the implementation of this for the CDL LINKER linkage editor. All of the macro templates and expansion directives are contained in the file MACROS.PGN and there are a few more macros in the 8080 implementation.

Differences Between 8080 and 6502 Pidgin

Discussion of these particular alterations is needed in order show the user how each fits the 8080/Z80 system under CP/M.

The parameter statement PARAMETER K$$=$$$ was designed because it was not possible to put all of the machine-dependent and system-dependent code in one area without it. As an example, the 6502 system for which Pidgin was designed uses 141 as an end-of-line newline mark, and the 8080/Z80 uses 13. End-of-file in the 6502 I/O system is defined as 255, and in the standard 8080/

```
LDA #0
TAX
```

which will save a byte. These work because Tincmp scans its list of macros from first to last. If the more specific macro is placed before the less specific, the more specific lines will be matched sooner. These optimizations might save ten bytes in several thousand. Bender made another optimization to get rid of an unusable jump instruction that is generated by the first CASE statement of a CHOOSE ON. This was possible through compiling code for an assembler – he could reset the ORG and obliterate the offending jump. Code generated for Maclod could also use this trick, but absolute code generated by Tincmp cannot.

I'm not sure I understand Bender's treatment of the information on closing a buffer, or his suggestion of separate buffers for each file. I have deliberately reused buffers, once a file has been completely read and closed, in order to save space. I believe this corresponds, however, to reassociating a new file name with the buffer, and it may not be what he is referring to. I have found it useful to put a statement to close a buffer immediately before each open statement. This obviously reflects a design problem, perhaps the same one. For me, quick on the reset button when a program isn't running right, the failure to close files on such a panic exit meant the program might not run when restarted because trying to open an opened buffer gives an error.

Bender's related closing of files before exiting is a good thing to automate and is possible in the framework of an operating system. But don't be lulled – a Pidgin program should close files before exiting anyhow. I had hoped not to assume an operating system for Pidgin, so that operating system software could be written in Pidgin. However, I'm glad to see a version that will support development of Pidgin programs in a friendly environment.

I am interested to see Bender's estimate that only a 20% gain in space could be achieved by handcoding. I have estimated that it might be 30-40%, but I have not done a study of it. The reason is that programs keep changing for a long time, and the ease of changing and porting a Pidgin program would be worth a factor of two in space – at least for programs no larger than 10K as compiled by Pidgin. As soon as a hand optimization was done, one would be very reluctant to change the program further. I believe Bender has made more use of run-time subroutines than I did. This achieves more compact code at a slight time penalty. In Spartan, I have made this tradeoff for compactness.

Both Bender and Anderson have made use of seven Pidgin programs which test the Pidgin macro definitions fairly thoroughly. These are available for the asking to anyone interested in implementing Pidgin.

Maclod and Spartan are available from Bootstrap Toolkit, 439 S. Orange Ave., S. Orange, NJ 07079. Write for prices and detailed descriptions of them, as well as of 8080 Tincmp, Pidgin and Meta4. The CP/M version contains the source code, assembly code and object files on an 8-inch, single-density CP/M format disk.

Z80 system it is 26. Putting these in a parameter statement makes all of these machine-dependent items easy to spot and change, increasing the portability of the program and clarifying its design. The 6502 Pidgin system can resolve these parameter references during the third pass and the 8080/Z80 system generates "EQU" statements to deal with the parameter statement. In order to properly process the assignment of parameter variables, the construction of the $$=+K$$ template was specifically designed to get the proper reference to a parameter variable. Thus, consider the statements shown here:

```
PARAMETER KLF=010
PARAMETER KNL=013
PARAMETER KEF=026
   :
   :
BEGINMAIN(AC,IAV)
   :
   :
EF=+KEF
LF=+KLF
NL=+KNL
   :
```

Assuming that NL,LF and EF were all declared BYTE, in the 8080/Z80 system the following code would be generated:

```
.KLF EQU 10
.KNL EQU 13
.KEF EQU 26
   :
   :
?START: LXI SP,?STACK
   CALL ?INIT
   EXTRN ?INIT
   :
   :
   :
   MVI A,.KEF
   STA .EF
   MVI A,.KLF
   STA .LF
   MVI A,.KNL
   STA .NL
   :
   :
```

It is necessary to comment at this point on the format of the assembly code generated by the 8080/Z80 version of Pidgin. In order to prevent the corruption of certain symbols which the assembler has self-declared as sacred and incorruptible (such as register names, and in the case of RMAC, the actual operation codes themselves), I have generated almost all labels with a leading period. This makes all declared symbols unique in that, for example, the casual use of "SP" as a byte variable in a Pidgin program won't blow the stack pointer assignment during the assembly of the resulting program. All subroutine references are generated with two leading periods so that a subroutine name could be unique and distinct from a variable name. As an example the Pidgin statement GOSUB ER would generate CALL ..ER. GOTO statements in Pidgin require label references elsewhere in the program. These references are always encoded with a leading question mark followed by the two letters "LL" which give a clue as to which statement generated the label. As an example, the statement GOTO 12 generates JMP ?LL012 and its label counterpart always generates ?LL102:. This also makes jump labels distinct from subroutine names. The informal syntax of Pidgin as stated in the article by Gale makes no restrictions on these labels so I have assumed them to be all unique. The WHILE statement always generates a .Xnnn: label which is the top of the loop statement. ENDWHILE generates a .Wnnn: label which identifies the bottom of the loop. Just prior to the .Wnnn: tag the compiler always compiles a jump instruction to the top of the loop. Within the WHILE . . . ENDWHILE sequence the ON statement generates a fetch and test sequence to jump to the bottom of the loop tag. Similarly, all the control sequences generate .iinnn: tags. An IF statement generates .Fnnn, CHOOSE ON generates no sequence, ENDCHOOSE generates an .XC sequence, and CASE generates a .UP sequence. These tags are important to the Pidgin programmer because in the case of unpaired statements, the assembly will produce undefined symbols in the area close to where the error occurred.

Whenever the BEGINMAIN directive is encountered, 8080/Z80 Pidgin always compiles two closed subroutines: ?SETAC and ?INTCK. ?SETAC is used by the comparison operators in setting the accumulator to one or zero depending on the outcome of the logical tests being performed and ?INTCK is used for integer inequality tests. A cell .ERASER. is also set aside for the CHOOSE ON/CASE statements. Since Tincmp is blind with regard to what has been happening overall in your program, it is difficult, without a second pass, to generate this code only if it is needed. After these standard sequences, the START label is generated and then the code to set the stack pointer. The subroutine ?INIT is then called. This subroutine may be coded to do anything the user wants done at initial entry to the Pidgin program. In this version of Pidgin (Version 1.05) it extracts the filenames from the user control statement and sets the global variable AC to the number of file names in the call line. Up to six names can be used in the call line. ?INIT then returns control to the user.

The TOP statement in 8080/Z80 Pidgin generates EXTRN statements for ?INIT, ?ENDING, ?MSG and ?BOOT labels. It then declares all of the system-wide global variables. Specifically, these are ER, the error indicator byte from the I/O System; IAV, an unused integer vector used to convey file information in the Apple II implementation; and AC, which contains the number of files mentioned in the user control statement. All of these values are generated in IOCTRL and labelled common. A "blank" common declaration is then generated so that all of the BYTE and INT declarations which follow it will be in "blank" common. This is done so that later versions of Pidgin will allow independently compiled external procedures. The subsequent occurrence of the BEGINMAIN statement then forces the end of the "blank" common region by the generation of a DSEG directive and a single word of DSEG memory ".ERASER.". This byte is then followed by the CSEG directive and the actual instructions generated by Pidgin.

The ENDMAIN directive generates a call to the closed subroutine "?ENDING" which in this version of Tincmp loops to close all open files. Return from ?ENDING is to a JMP ?BOOT which reboots from zero in CP/M. BOTTOM generates a stack of length 60 and an END ?START directive. RETURN and ENDSUB each generate RET instructions. There is no check for unbalanced SUB . . . ENDSUB directives or for RETURN in a main program segment.

There is a special template $$=+000 which is used to generate the XRA A, STA .$$ code sequence. This eliminates one byte from the generated code. Since the 6502 has no "zero address" or immediate address operands, these special cases are not included in the original distribution.

Although discussed more fully

below, the statement ASSOCIATE FCB $ WITH I$$ is used to attach the *n*th file in the control statement to a particular file number contained in an integer variable. Admittedly somewhat peculiar, this statement was contrived in an attempt to reduce the operating system interface problems between the Apple II 6502 implementation and the 8080/Z80 CP/M implementation. While *not* implemented, ASSOCIATE $$ WITH I$$ is the more general form of this statement. In this statement, $$ is the first of twelve byte locations. This location contains a blank if the default drive is to be used for the subsequently named file or the letter corresponding to a drive identifier.. The next eleven bytes contain the full, unambiguous file name of the file to be used. I$$ should contain a file number in the low order seven bits.

I/O Control System Description

The I/O Control System for Pidgin on the 8080/Z80 (PIOCS) is designed to run under CP/M 2.2 or 1.4. It will handle all of the usual CP/M file conventions and can access any sequential file written by CP/M. With some modifications it could do random file I/O as well. For the purpose of this discussion, the routines referred to are all in the element "IOCMPLX.ASM" and "SYSTEM.ASM". Note: the two routines READBUF and WRITEBUF are not implemented in this release of Pidgin, since they were not used in Tincmp.

The first routine to discuss is ?INIT. This subroutine gains control from the main program at the invocation of the BEGINMAIN statement. ?INIT examines the temporary buffer which has been set by CP/M to contain the processor call image. If there are characters in this image then ?INIT will scan each file name in the image, constructing an FCB (File Control Block) for each filename found. Even if you do not use these files, the FCB's are constructed. As each FCB is built, an area of memory called ?IOCTL is also initialized. This area consists of six seven-word entries, one for each possible open file in the system. The first word in this seven-word entry is the operation to be performed on each file. This currently is read (bit zero=1) or write (bit one=1) and a state called prepare (bit two=1). The prepared state means that the Pidgin program has established an association between the filename and an internally assigned file number. Currently, in order to use Pidgin, you need to assign all of the files in the processor call, but that need not be the only way to associate files with file numbers. The prepared bit is bit two=1. When the ASSOCIATE FCB n WITH Ixx is executed, the FCB number n in the processor call line is equated in the ?IOCTL table with file number Ixx. In order to keep compatible with other versions of Pidgin, Ixx is used but 8080/Z80 Pidgin only looks at the low order seven bits of Ixx. This means that file numbers which do not have unique bits 0-6 are considered the same file by this I/O package. After ?INIT sets up the I/O tables it exits back to the users.

?ATACH is the routine invoked by the ASSOCIATE statement. It inserts the proper information into the ?IOCTL table and marks the file "prepared" so that it many now be properly referenced by number of buffer, as described in the 6502 Pidgin description and the 6502 file system description (see *DDJ* #57).

?IOPEN is called when the user executes an OPEN statement. At this time, the read/write bit is set in ?IOCTL and the prepared bit is cleared. The buffer address for the user buffer is stored in ?IOCTL as well as the FCB address and the file number. ?IOPEN then calls the resident file system to open a read file or delete and make a write file. The buffer counter is set to 128 in the case of a read file, or to zero in case of a write file. This will cause read to read a buffer full of data as soon as it is entered for the first time; write will assume that its buffer is empty.

?BREAD is called to read a single character. It examines the ?IOCTL table, looking for a read file with the proper buffer address, and then looks at the buffer counter. If less than 128 characters have been transferred from the read buffer, ?BREAD gets the next character in turn and gives it to the user. If the buffer counter is equal to 128 then a read is initiated through the resident file system using the FCB assigned to the buffer. If an end file is reached, ?BREAD returns a standard CP/M end-of-file character, which is 26 rather than 225. ER is set to 1 if an end-of-file is detected. Note that the file is *not* closed on an end-of-file, and it is certainly possible to keep issuing reads on a file beyond the end-of-file.

?BWRIT is similar to ?BREAD. The character to be written is added to the buffer, which is written out in blocks of 128 bytes as each buffer is filled. When the file is closed (see below), the buffer is padded out with all end-of-file characters (26's) and the final block is written to disk.

?BCLOF is a close file routine. This routine *destroys* all of the information in ?IOCTL so that it is no longer possible to read, write, or even to open the file associated with the buffer mentioned in a close statement. *For this reason I strongly urge the use of separate buffers for each file open at the same time, considering this a bad structural weakness in Pidgin.* ?BCLOF flushes the write buffer before it closes it out. The previous entry in ?IOCTL is made re-useable by other calls to ASSOCIATE, OPEN, etc.

?ENDING is the last routine in the PIOCS. When it is invoked by ENDMAIN, it examines the information in ?IOCTL and clears all of the open write buffers by "flushing" them. The write files are then closed by calls to CP/M. Similarly, each read file found in the ?IOCTL table causes a call to CP/M to close it. ?ENDING then returns control to the Pidgin program, executing the next statement, which is a jump to the ?BOOT entry point.

The entries to CP/M are defined in SYSTEM.ASM so that you can use Pidgin under MP/M. No address is hard-coded except in the SYSTEM element. Also in SYSTEM is the load DE register from the address in HL.

?WRNXT is in WRNXT.ASM along with ?RNXT. These routines write and read the console device. ?ISTOP is also in WRNXT and is used to display the message "STOP STATEMENT n ENCOUNTERED IN SOURCE PROGRAM" where n is the stop number 0 through 9. This, of course, could be changed to anything else you desire if it is not satisfactory.

The Arithmetic Package

Certain integer functions in Pidgin cannot be performed directly in 8080/Z80 hardware. For example, multiplication and division are not hardware functions unless the multiplier is a power of two or the divisor is a power of two, in which case the option of using left- and right-shift instructions is available. In these cases and others as well, closed subroutines are provided to perform the functions not provided in hardware. All of these subroutines are provided in the relocatable element PIDMATH.ASM. This element was written by Richard Curtiss and is provided here with his permission.

Certain functions are also provided which allow you to do other arithmetic functions, such as modulus arithmetic, signed and unsigned arithemetic and the like, but these functions are not essential for use in Pidgin programs and are not directly called by Pidgin. Nevertheless, they remain in the package for your use.

Integer subtraction is provided directly by the subtract instruction for unsigned eight-bit quantities, but for the 8080 processor, a double-precision subtract instruction is not provided in the hardware so the routine ?ISUB in the math package is used. In the Z80 there is a direct subtract instruction, but it subtracts with carry, meaning that the carry bit needs to be cleared prior to subtraction. This could be directly implemented in the hardware by performing an ORA A first, but since Pidgin was written for an 8080, the ?ISUB routine was used. For multiplication, ?IMUL is used and for division ?IDIV is used. An outline of the closed subroutine arithmetic routines is given in the assembly language listing of Pidmath.

Certain comparison functions require that some arithmetic be performed and then a logical indicator set to the result of the comparison. ?CDEHL compares the values in the DE and HL registers. In certain instances, the relation between the two values compared is tested. If the relation is true, code is executed to return a low-order one bit, and if the relationship is false, a zero is returned. Some of this code is generated in-line in the Pidgin program. Where the code would be complex, out-of-line subroutines are included.

Logical operations performed on bytes are always performed by in-line code, and the code generated always returns either a low-order one bit or a zero bit. At the conclusion of any logical operation, the left seven bits are always zero. When a logical test is to be performed, the operand of the test is loaded into the accumulator and an ORA A instruction is executed to set the proper flag bits to zero or non-zero. Zero always implies that the logical value was false, and non-zero always means true. Because only one bit is used, operations on this bit always influence the logical meaning of the logical expression.

Getting Pidgin Running

There are two ways to get Pidgin up and running on your computer. The easiest way is to use the COM file Tincmp, the macro file MACROS.PGN and your source code in Pidgin to produce a file of assembly code. This method can be used for any machine on which CP/M is running and the TPA (Transient Program Area) starts at 100H. You will need about 15 Kilobytes of free memory to hold the Tincmp compiler, its tables and subroutines. This method may not be suitable for all readers and some may have to begin generation of Tincmp at another start address or desire some different features. In any case, you can start over with STAGE2 or use the assembly language output by Tincmp during its own operation. By hand coding the file TINCMP.MAC you could save about 20% of the memory required for the Tincmp program, but this is such a small area that it won't buy much. Because the I/O lights on the disk file are on all the time during operation of the Tincmp, I don't think you will save much time by hand optimizing the Tincmp compiler.

If you want to include operations for a Z80 processor, I suggest the following: use the Tincmp compiler as supplied here to process a new file of MACROS.PGN. These macros could be written to take advantage of the Z80 instruction set. When you have a working set of macros, recompile Tincmp and test it. When you are sure that you have a working copy of Tincmp, replace the 8080 version with the Z80 version. Then, work your way through the I/O complex, recoding that for the Z80. I wrote the one supplied here with the Z80 in mind. Later, change the other routines as you have more time. I don't believe that any changes will give a significant increase in execution speed, since many Z80 instructions are just as long and slow as the multiple 8080 instructions that they replace. As an example, consider the load DE from memory in the Z80. In the 8080 you need a load HL from memory and an exchange. Assuming that HL need not be saved prior to this use, and in my macro set there are no examples of this at this writing, each instruction is four bytes long and the 8080 pair of instructions take 20 T cycles, just as the Z80 single instruction does. Since the 8085 is a superset of the 8080 instruction set, the instructions need not be changed to make use of the 8085 processor.

If you have a copy of the CDL LINKER program (Computer Design Labs of Trenton, NJ) you *can* have your Tincmp compiler turn out direct code *without* assembly. You will need three passes through Tincmp. Your macros will be structurally similar to Bill Gale's set for the 6502. The LINKER program, unlike the Microsoft Link 80 routine or the Digital Research Link routine, accepts hexadecimal as input and generates linkage edited absolute modules from this input. You will have to modify the Pidgin library routines so that the CDL MACRO2 assembler can assemble them and use them as library routines, because the CDL LINKER cannot accept the Microsoft relocation format used by both Microsoft and Digital Research. The net result is that your program will compile faster with the three passes through Tincmp than it would going through Tincmp once and then being assembled. Assemblers which are powerful are slow. MACRO80 and RMAC are not exceptions to this rule. MACRO2, although somewhat faster than both RMAC and MACRO80, still requires more time than the additional two passes through Tincmp would. The negative side of this three pass compilation is that the resulting object code is difficult to "clean up" or hand optimize. In addition, the macros are far more complex than in the one-pass compilation.

Availability

The 8080/Z80 version of Tincmp operating under CP/M is available from the author on eight-inch, single-density floppy disk in CP/M format for $25. It is also available from the SIG/M group of the Amateur Computer Group of New Jersey at a more reasonable price. No matter where you purchase the disk, please note that all of the programs on the disk are copyrighted. Gale, Curtiss and Bender all hold copyrights on the programs on the disk, and allow the owner of the disk to make as many copies as desired for non-commercial, personal use. Commercial use is forbidden without the written consent of each of the authors. No copies may be made for commercial use.

The disk contains the following: STAGE2 written in FLUB as a COM file. STAGE2 macros for Pidgin, Tincmp written in Pidgin. A COM file of Tincmp. Pidgin macros for Tincmp. The library routines for Pidgin. This document and a short guide to getting Pidgin up on other assemblers. After 1982, if you are writing to me (directly) for a copy of the disk,

inquire first. I will be happy to answer any questions you might have concerning Tincmp or Pidgin, but please write them out and enclose a self-addressed, stamped envelope.

Acknowledgements

The author acknowledges the help of Dr. William Gale, the originator of Pidgin, who made many valuable suggestions during this work. I also acknowledge the groundwork done by Richard Curtiss in his development of a super implementation of STAGE2 for the 8080 processor and for his kind permission to use all of the routines included in his CP/MUG library distribution. ▶▶J

Bibliography

Gale, W. A.: "Pidgin: A System Programming Language." *Dr. Dobb's Journal* 6: 10 #57, July 81.

Gale, W. A.: "A Tiny File System." *Dr. Dobb's Journal* 6:20 #56, June 81.

Waite, W. M.: *Implementing Software for Non-Numeric Applications.* Prentice-Hall, Inc. Englewood Cliffs, NJ, 1973.

Curtiss, R.: "STAGE2, FLUB and associated materials." *CP/MUG Library Disk* #51. Order from CP/M Users Group NY, NY. Current cost about $8.

CP/M, LINK-80, RMAC, PL/I-80 and MP/M are registered trademarks of Digital Research Inc., Pacific Grove, CA. Z80 is a trademark of Zilog Division of Exxon, Inc., Cupertino, CA.

The author has been involved in computer programming for the past 21 years. He practices medicine, specializes in neurology and likes kids, cats and Bach although not necessarily in that order.

(Listing begins on page 40)

Listing I

Listing 1. This program, written in Heath's Benton Harbor BASIC, provides a run-through of all possible combinations of LED patterns.

```
00100 A=8203:FOR I=0TO 9:POKE A+I,255:NEXT I:REM Clear the display
00110 CNTRL 2,1:REM Turn H-8 display on without update to write to the LEDs
00120 PRINT "This program demonstrates the use of the H-8 front panel LEDs."
00130 PRINT "To view a run-through of all possible combinations, enter '1'."
00140 PRINT "To select particular displays, enter '2'."
00150 PRINT "To halt the run at a given display, type 'CTRL-B'"
00160 PRINT "To exit program 2, enter '-1.'"
00170 INPUT "Enter your choice: ";A
00180 IF A=1THEN 210
00190 IF A=2THEN 340
00200 PRINT :PRINT "Illegal option.  Re-enter.":PRINT :GOTO 130
00210 CNTRL 0,310:REM When CTRL-B is pressed, goto 310
00220 A=8203:B=8204:C=8205:D=8206:E=8207:F=8208:REM Set initial LED locations
00230 PRINT :PRINT "Decimal # corresponding to LED:"
00240 FOR I=1TO 257:REM Full circle on the patterns
00250 POKE A,I:POKE B,I:POKE C,I:POKE D,I:POKE E,I:POKE F,I
00260 PRINT I;:REM Show what number was just displayed
00270 PAUSE 200:REM Wait a bit so user can see pattern
00280 PRINT CHR$(13);:REM CR without LF:REM To keep display from scrolling off
00290 NEXT I
00300 GOTO 120
00310 PRINT "The display corresponds to";I;"; press RETURN to continue."
00320 PAUSE :REM Wait for a RETURN to be entered to continue
00330 GOTO 290
00340 A=8203:B=8204:C=8205:D=8206:E=8207:F=8208
00350 PRINT "Enter the number to be displayed (-1 to end): ";
00360 INPUT "";I
00370 IF I=-1THEN 120
00380 POKE A,I:POKE B,I:POKE C,I:POKE D,I:POKE E,I:POKE F,I:REM Same for all LED
```

```
s
00390 GOTO 360
00400 END
```

End Listing I

Listing II

Listing 2. This program prints a simple message on the front panel LED display of the H8 computer.

```
100 PRINT "LED PROGRAMMING DEMONSTRATION":PRINT
110 PRINT "To operate, enter data starting at line #500.  Enter the number of"
120 PRINT "data entries at line #480."
130 PRINT :PRINT "To run the program with existing data, press 'RETURN.'":PAUSE
140 REM Turn on LED's without update
150 CNTRL 2,1
160 REM Port number for left-most LED
170 A=8203
180 REM Erase LED
190 FOR I=0TO 8:POKE A+I,255:NEXT I
200 REM Read number of characters to print
210 READ N
220 FOR K=1TO N
230 REM Read data for each character
240 READ L
250 REM Write each character, then erase (right to left)
260 POKE A+8,L:POKE A+7,L
270 POKE A+8,255
280 PAUSE 20:REM Wait a bit so user can read the display
290 POKE A+7,255:POKE A+6,L
300 PAUSE 20
310 POKE A+6,255:POKE A+5,L
320 PAUSE 20
330 POKE A+5,255:POKE A+4,L
340 PAUSE 20
350 POKE A+4,255:POKE A+3,L
360 PAUSE 20
370 POKE A+3,255:POKE A+2,L
380 PAUSE 20
```

(Continued on page 22)

Pattern Classification

(Text begins on page 23)

```
0010 REM: A BASIC PROGRAM TO DETERMINE FISHER'S LINEAR DISCRIMINANT
0020 REM
0030 REM
0040 REM
0050 REM
0060 REM: PRESENT REV: 1.04, AUGUST 31, 1981.
0070 ON ESC THEN GOTO 2930
0080 CLOSE
0090 ON ERR THEN GOTO 0110
0100 DELETE "PLOTDAT"
0110 ON ERR THEN STOP
0120 PAGE =80
0130 GOSUB 2990
0140 PRINT
0150 PRINT "FISHER'S LINEAR DISCRIMINANT."
0160 PRINT
0170 PRINT " PROGRAM OPTIONS:"
0180 PRINT
0190 PRINT "    1 - BUILD, ADD TO, OR EDIT AN EXISTING TRAINING DATA FILE."
0200 PRINT
0210 PRINT "    2 - COMPUTE THE FISHER'S LINEAR DISCRIMINANT FOR AN EXISTING ";
0220 PRINT "        TRAINING DATA FILE. (ALLOWS SELECTION OF UNCLASSIFIED POINTS
),"
0230 PRINT
0240 PRINT "    3 - PRINT AN EXISTING TRAINING DATA FILE."
0250 PRINT
0260 INPUT " NUMBER OF THE DESIRED PROGRAM OPTION ? ",I
0270 ON ESC THEN GOTO 0070
0280 IF I<1 THEN GOTO 0260
0290 IF I>3 THEN GOTO 0260
0300 DIM N$(13),A$(1),A1$(26),C(1)
0310 PRINT
0320 INPUT "WHAT IS YOUR TRAINING DATA FILE NAME? ",N$
0330 PRINT
0340 INPUT "HOW MANY FEATURES IN EACH SAMPLE?(MAX=25) ",N2
0350 IF N2<1 THEN GOTO 0340
0360 IF N>25 THEN GOTO 0340
0370 DIM X(N2),T(2,N2),M1(N2),M2(N2),U1(N2),U2(1)
0380 IF I=2 THEN GOTO 0440
0390 IF I=1 THEN GOTO 0420
0400 GOSUB 3740
0410 GOTO 2930
0420 GOSUB 3320
0430 GOTO 2930
0440 PRINT
0450 LET U=0
0460 LET R=0
0470 INPUT "DO YOU WISH TO PLOT AN UNCLASSIFIED DATA POINT? ",A$
0480 PRINT
0490 IF A$="Y" THEN GOSUB 3020
0500 ON ESC THEN GOTO 2930
0510 PRINT "THANKYOU. THE FISHER'S LINEAR DISCRIMINANT IS BEING COMPUTED."
0520 OPEN FILE (0,0),N$
0530 REM: COMPUTE THE UNPROJECTED SAMPLE VECTOR MEANS.
0540 LET I=0
0550 LET J=0
0560 LET K=0
0570 MAT READ FILE (0,I),C,X
0580 LET C=C(1)
0590 IF EOF(0) THEN GOTO 0760
0600 IF C=1 THEN GOTO 0630
0610 IF C=2 THEN GOTO 0690
0620 GOTO 2810
0630 REM: CLASS 1 TOTALIZATION.
0640 FOR L=1 TO N2
0650   LET T(1,L)=T(1,L)+X(L)
0660 NEXT L
0670 LET J=J+1
0680 GOTO 0740
0690 REM: CLASS 2 TOTALIZATION.
0700 FOR L=1 TO N2
0710   LET T(2,L)=T(2,L)+X(L)
0720 NEXT L
0730 LET K=K+1
0740 LET I=I+1
0750 GOTO 0570
0760 IF J=0 THEN GOTO 2850
0770 IF K=0 THEN GOTO 2850
0780 FOR L=1 TO N2
0790   LET M1(L)=T(1,L)/J
0800   LET M2(L)=T(2,L)/K
0810 NEXT L
0820 LET N=J+K
0830 LET P=N2
0840 REM: COMPUTE SCATTER MATRICES.
0850 DIM S1(P,P),S2(P,P),W1(P,P),D(P),D1(P),D2(P,P)
0860 MAT S1=ZER(P,P)
0870 MAT S2=ZER(P,P)
0880 FOR I=0 TO N-1
0890   MAT READ FILE (0,I),C,X
0900   LET C=C(1)
0910   IF C=2 THEN GOTO 0970
0920   MAT D=X-M1
0930   MAT D1=TRN(D)
0940   MAT D2=D*D1
0950   MAT S1=S1+D2
0960   GOTO 1010
0970   MAT D=X-M2
0980   MAT D1=TRN(D)
0990   MAT D2=D*D1
1000   MAT S2=S2+D2
1010 NEXT I
1020 MAT W1=S1+S2
1030 REM: COMPUTE THE WITHIN CLASS SCATTER MATRIX INVERSE.
1040 DIM V(P,P),M3(25),X1(25)
1050 ON ERR THEN GOTO 2870
1060 MAT V=INV(W1)
1070 ON ERR THEN STOP
1080 REM: COMPUTE THE WEIGHTING VECTOR W.
1090 DIM W(P)
1100 MAT M3=M1-M2
1110 MAT W=V*M3
1120 REM: COMPUTE THE SCALAR PROJECTED POINTS.
1130 DIM Y(1),Y1(J),Y2(K)
1140 LET J1=0
1150 LET K1=0
1160 MAT X1=TRN(W)
1170 FOR I=0 TO N-1
1180   MAT READ FILE (0,I),C,X
1190   LET C=C(1)
1200   IF C=2 THEN GOTO 1250
1210   LET J1=J1+1
1220   MAT Y=X1*X
1230   LET Y1(J1)=Y(1)
1240   GOTO 1280
1250   LET K1=K1+1
1260   MAT Y=X1*X
1270   LET Y2(K1)=Y(1)
1280 NEXT I
1290 IF U=0 THEN GOTO 1330
1300 MAT U2=X1*U1
1310 LET U2=U2(1)
1320 IF R=1 THEN GOTO 1720
1330 PRINT
1340 INPUT "DO YOU WISH TO SEE A LISTING OF THE PROJECTED POINTS? ",A$
1350 IF A$="Y" THEN GOTO 1370
1360 GOTO 1720
1370 LET L=0
1380 LET L7=0
1390 LET L8=0
1400 GOSUB 2990
1410 PRINT "                          PROJECTED POINTS FOR:"
1420 PRINT
1430 PRINT "                  CLASS 1   ";"             ";"  CLASS 2  "
1440 NOESC
1450 FOR I=1 TO 18
1460   LET L=L+1
1470   LET L7=L7+1
1480   PRINT "                ";
1490   IF L7>J THEN GOTO 1520
1500   PRINT USING "-----#.######",Y1(L);"   ";
1510   GOTO 1530
1520   PRINT "        ";
1530   LET L8=L8+1
1540   IF L8>K THEN GOTO 1560
1550   PRINT USING "-----#.######",Y2(L);
1560   PRINT
1570 NEXT I
1580 ESC
1590 ON ESC THEN GOTO 2930
1600 PRINT
1610 INPUT "PRESS RETURN TO CONTINUE.",A$
1620 IF L>J THEN GOTO 1640
1630 GOTO 1400
```

Pattern Classification

(Listing continued)

```
1640 IF L>K THEN GOTO 1660
1650 GOTO 1400
1660 IF U=0 THEN GOTO 1720
1670 PRINT

1680 PRINT "PROJECTION OF THE UNCLASSIFIED POINT=";
1690 PRINT USING "-----#.######",U2
1700 PRINT
1710 INPUT "PRESS RETURN TO CONTINUE.",A$
1720 REM: PLOT THE PROJECTED DISTRIBUTION.

1730 GOSUB 2990
1740 PRINT "ONE MOMENT PLEASE."
1750 PRINT "THE PLOT MATRIX IS BEING COMPUTED."
1760 IF R=1 THEN GOTO 1960
1770 REM: SCALE THE PROJECTED POINTS: (0<=Y<=70).
1780 LET L1=Y1(1)

1790 LET L2=Y1(1)
1800 FOR I=2 TO J
1810   IF Y1(I)<L1 THEN LET L1=Y1(I)
1820   IF Y1(I)>L2 THEN LET L2=Y1(I)

1830 NEXT I
1840 FOR I=1 TO K
1850   IF Y2(I)<L1 THEN LET L1=Y2(I)
1860   IF Y2(I)>L2 THEN LET L2=Y2(I)

1870 NEXT I
1880 LET L3=L2-L1
1890 DIM Y3(J),Y4(K)
1900 FOR I=1 TO J
1910   LET Y3(I)=INT((((Y1(I)-L1)/L3)*70)+.5)
1920 NEXT I
1930 FOR I=1 TO K
1940   LET Y4(I)=INT((((Y2(I)-L1)/L3)*70)+.5)
1950 NEXT I
1960 LET E=0
1970 IF U=0 THEN GOTO 2080
1980 LET U3=INT((((U2-L1)/L3)*70)+.5)
1990 IF U3<0 THEN GOTO 2010
2000 GOTO 2030
2010 LET U3=0
2020 GOTO 2060
2030 IF U3>70 THEN GOTO 2050
2040 GOTO 2070
2050 LET U3=70
2060 LET E=1
2070 IF R=1 THEN GOTO 2280
2080 REM: BUILD FISHER PLOT FILE.
2090 LET R1=0
2100 LET L6=0
2110 DIM L$(80)
2120 IF R=1 THEN GOTO 2150
2130 OPEN FILE (2,0),"PLOTDAT"
2140 DIM Z(N)
2150 FOR I=1 TO J
2160   LET Z(I)=Y3(I)
2170 NEXT I
2180 LET K7=J
2190 LET P$="1"
2200 GOSUB 3150
2210 FOR I=1 TO K
2220   LET Z(I)=Y4(I)
2230 NEXT I
2240 LET K7=K
2250 LET P$="2"
2260 GOSUB 3150
2270 IF U=0 THEN GOTO 2380
2280 LET Z(1)=U3
2290 LET R1=R1+1
2300 IF R1<27 THEN GOTO 2320
2310 LET R1=R1-26
2320 LET A1$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2330 LET P$=A1$(R1,R1)
2340 IF E=0 THEN GOTO 2360
2350 LET P$=" "
2360 LET K7=1
2370 GOSUB 3150
2380 REM: PLOT THE PROJECTED POINTS.
2390 GOSUB 2990
2400 IF L6>12 THEN GOTO 2440
2410 FOR I=1 TO 13-L6
2420   PRINT
2430 NEXT I
2440 PRINT "                          FISHER LINEAR DISCRIMINANT PLOT"
2450 PRINT
2460 FOR I=0 TO L6
2470   READ FILE (2,L6-I),L$
2480   PRINT "    ";L$(1,72)
2490 NEXT I
2500 PRINT "    !";
2510 FOR I=1 TO 14
2520   PRINT "++++!";
2530 NEXT I
2540 PRINT "    !
      !"
2550 FOR I=0 TO 2
2560   LET L9=L1+(L3*I/2)
2570   PRINT USING "---#.####",L9;
2580   IF I=2 THEN GOTO 2600
2590   PRINT "                          ";
2600 NEXT I
2610 PRINT
2620 PRINT
2630 PRINT "SCALE FACTOR: EACH SPACE= ";
2640 PRINT USING "---#.####",L3/70;"     ";
2650 IF U=0 THEN GOTO 2780
2660 PRINT "UNCLASSIFIED POINT ";P$;"=";
2670 PRINT USING "-----#.######",U2
2680 IF E=0 THEN GOTO 2700
2690 PRINT "                                                (EXCEEDS PLOT BOUNDRIES S
HOWN)"
2700 INPUT "DO YOU WISH TO PLOT ANOTHER UNCLASSIFIED POINT? ",A$
2710 IF A$="Y" THEN GOTO 2730
2720 GOTO 2800
2730 GOSUB 2990
2740 GOSUB 3020
2750 ON ESC THEN GOTO 2930
2760 LET R=1
2770 GOTO 1300
2780 INPUT "DO YOU WISH TO PLOT AN UNCLASSIFIED POINT? ",A$
2790 IF A$="Y" THEN GOTO 2730
2800 GOTO 2930
2810 PRINT
2820 PRINT "A DATA MISCLASSIFICATION HAS OCCURED. PLEASE"
2830 PRINT "CORRECT THE PROBLEM IN THE DATA BASE AND TRY AGAIN."
2840 GOTO 2930
2850 PRINT "EITHER CLASS 1 OR CLASS 2 HAS NO ELEMENTS. THEREFORE, CLASSIFICATION
 IS MEANINGLESS."
2860 GOTO 2930
2870 ON ERR THEN STOP
2880 GOSUB 2990
2890 PRINT
2900 PRINT "THE WITHIN CLASS SCATTER MATRIX IS SINGULAR. THEREFORE, "
2910 PRINT "THE CURRENT DATA CANNOT BE USED IN THIS PROGRAM."
2920 PRINT "EXECUTION IS ABORTED !!!"
2930 CLOSE
2940 ON ERR THEN GOTO 2960
2950 DELETE "PLOTDAT"

2960 ON ERR THEN STOP
2970 PAGE =72
2980 END
2990 REM: SUBROUTINE TO FORM FEED THE 1420 CRT.
3000 LWRITE "↙<28>"
3010 RETURN
3020 REM: SUBROUTINE TO INPUT FISHER DATA.
3030 ON ESC THEN GOTO 3120
3040 LET U=1
3050 PRINT
3060 LET I=1
3070 PRINT "COORDINATE #";I;"? ";
3080 INPUT "",U1(I)
3090 LET I=I+1
3100 IF I>N2 THEN GOTO 3130
3110 GOTO 3070
3120 LET U=0
3130 PRINT
3140 RETURN
3150 REM: SUBROUTINE TO COMPUTE THE PLOT MATRIX.
3160 LET I=1
3170 LET I1=0
3180 READ FILE (2,I1),L$
3190 IF L$="" THEN GOSUB 3300
3200 IF L$(Z(I)+1,Z(I)+1)=" " THEN GOTO 3250
```

```
3210 LET L$=""
3220 LET I1=I1+1
3230 IF I1>L6 THEN LET L6=I1
3240 GOTO 3180
3250 LET L$(Z(I)+1,Z(I)+1)=P$
3260 WRITE FILE (2,I1),L$
3270 LET I=I+1
3280 IF I>K7 THEN GOTO 3310
3290 GOTO 3170
3300 LET L$="
            "
3310 RETURN
3320 REM:SUBROUTINE TO BUILD A FISHER DATA BASE FROM TRAINING DATA.
3330 DIM F(N2)
3340 OPEN FILE (0,0),N$
3350 GOSUB 2990
3360 PRINT
3370 ON ESC THEN GOTO 3730
3380 INPUT "BEGINNING (OR NEXT) SAMPLE (RECORD) NUMBER? ",I
3390 ON ESC THEN GOTO 3350
3400 GOSUB 2990
3410 IF I=0 THEN GOTO 3440
3420 MAT READ FILE (0,I-1),C,X
3430 IF EOF(0) THEN GOTO 3360
3440 PRINT "RECORD #: ";I
3450 PRINT
3460 LET C(1)=0
3470 MAT READ FILE (0,I),C,X
3480 IF C(1)=0 THEN GOTO 3580
3490 PRINT "CURRENT CONTENTS OF RECORD ";I;":"
3500 PRINT "CLASS= ";C(1)
3510 PRINT "FEATURES:"
3520 FOR J=1 TO N2
3530   PRINT USING "----#.######",X(J);
3540   PRINT " ";

3550 NEXT J
3560 PRINT
3570 GOTO 3590
3580 PRINT "THIS ENTRY IS CURRENTLY EMPTY."
3590 PRINT
3600 MAT INPUT "CLASS (1 OR 2)? ",C
3610 IF C(1)<1 THEN GOTO 3600
3620 IF C(1)>2 THEN GOTO 3600
3630 PRINT
3640 LET J=1
3650 PRINT "FEATURE #";J;" ? ";
3660 INPUT "",F(J)
3670 LET J=J+1
3680 IF J>N2 THEN GOTO 3700
3690 GOTO 3650
3700 MAT WRITE FILE (0,I),C,F
3710 LET I=I+1
3720 GOTO 3400
3730 RETURN
3740 REM: SUBROUTINE TO PRINT TRAINING DATA BASE.
3750 ON ESC THEN GOTO 3980
3760 GOSUB 2990
3770 OPEN FILE (0,0),N$
3780 PRINT "DATA BASE: ";N$
3790 PRINT
3800 PRINT "RECORD#  CLASS:  FEATURES:"
3810 LET I=0
3820 MAT READ FILE (0,I),C,X
3830 IF EOF(0) THEN GOTO 3980
3840 IF I>9 THEN GOTO 3860
3850 PRINT " ";
3860 PRINT I;"      ";C(1);"   ";
3870 LET J=1
3880 PRINT USING " --#.###",X(J);
3890 IF J-((INT(J/7))*7)>.01 THEN GOTO 3920
3900 PRINT
3910 PRINT "                ";
3920 LET J=J+1
3930 IF J>N2 THEN GOTO 3950
3940 GOTO 3880
3950 PRINT
3960 LET I=I+1
3970 GOTO 3820
3980 ON ESC THEN GOTO 4010
3990 PRINT
4000 INPUT "PRESS RETURN TO CONTINUE.",A$
4010 RETURN
```

End Listing

Listing I

```
TOP$ TINCMP COPYRIGHT (C) 1981 W.A.GALE
PARAMETER KLF=010$ CP/M MODIFICATION AND 8080 RECODING
PARAMETER KCR=013$ BY A. L. BENDER, M. D.
PARAMETER KEF=026$ NEW MODS AND REWORKING COPYRIGHT (C) 1981 A L BENDER, M D
BYTE AA$ WORK
BYTE BB$ WORK BYTE
BYTE DD$ WORK
BYTE EE$ WORK BYTE

BYTE BF(080)$ EXPANSION BUFFER
BYTE BL$ BLANK
BYTE BP$ POINTER INTO BF
BYTE C0$ CONSTANT ZERO
BYTE C1$ CONSTANT ONE
BYTE C2$ CONSTANT TWO
BYTE C3$ CONSTANT 3
BYTE C4$ CONSTANT  040
BYTE C8$ CONSTANT 080
BYTE C9$ CONSTANT 9
BYTE CC$ INPUT CHARACTER
BYTE CX$ CONSTANT TEN

BYTE DG$ DIGIT FROM PARAMTER TREATMENT DEFINITION
BYTE DS(010)$ DIGIT STACK FOR SUB SD
BYTE EF$ END FILE CHARACTER
BYTE F1(00128)$ INPUT BUFFER
BYTE F2(00128)$ OUTPUT BUFFER
BYTE HA$ 'A'
BYTE HF$ 'F'
BYTE LE$ END OF LIST
BYTE LF$ LINE FEED CHARACTER
BYTE LS(09000)$ LIST OF MACRO DEFINITIONS
BYTE MF$ MACRO REPLACEMENT OPERATOR FLAG
BYTE ML$ MACRO LENGTH
BYTE MM$ MINIMUM MACRO LENGTH

BYTE ND$ NUMBER OF DIGITS USED IN SUB SD FOR NUMBER OUTPUT
BYTE NL$ NEW LINE
BYTE O1$ FETCH CODE
BYTE O2$ INDEX CODE
BYTE O3$ DISPOSE CODE
BYTE OA$ '+' ADD OPERATOR
BYTE OB$ '!' POP STACK OPERATOR
BYTE OC$ 'C' CHARACTER DISPOSE
BYTE OD$ 'V' DIGIT CONVERSION FETCH
BYTE OE$ ESCAPE CHARACTER
BYTE OG$ IGNORE CHARACTER
BYTE OH$ 'H' HEX CONVERSION FETCH
BYTE OL$ 'L' LITERAL FETCH
BYTE OM$ '*' MULTIPLY DISPOSE
BYTE ON$ 'N' NUMERIC LITERAL FETCH

BYTE OP$ 'P' PARAMETER FETCH OR DISPOSE
BYTE OR$ '-' REDUCE (SUBTRACT) DISPOSE
BYTE OS$ 'S' STACK FETCH OR DISPOSE

BYTE OT$ TRACE FLAG TURN ON
BYTE PP$ POINTER INTO IPR
BYTE RB$ BEGIN DEFINITION FLAG
```

```
BYTE RC; (COMMENT) END OF LINE FLAG
BYTE SF; SUBSTITUTION PARAMETER FLAG
BYTE SP; STACK POINTER
BYTE TR; TRUE IF NO TRACE
BYTE UG; USE IGNORE;  TRUE UNLESS OG IS 'X'
BYTE UN; NOT X-- FLAG FOR NOT SUPPRESSING NEW LINES ON OUTPUT
BYTE UO; USE OPERATIONS-- TRUE UNLESS MF IS 'X'
BYTE UT; USE TRACE  TRACE MODE IS ON
BYTE ZR; CHARACTER ZERO
INT I00; CONSTANT ZERO
INT I01; CONSTANT 1
INT I09; CONSTANT 9
INT I10; CONSTANT 10
INT I16; CONSTANT 16
INT IAA; WORK
INT IBB;  WORKING STORAGE
INT IBC; BUCKET NUMBER
INT IDP; DEFINITION POINTER WHILE MATCH
INT IED; POINTS TO END OF DEFINITIONS
INT III; POINTER TO L WHILE READING
INT IJJ; POINTER TO L READING CODE
INT ILM; MAXIMUM LIMIT FOR STORING IN L

INT ILP(01000); POINTERS TO MACROS
INT IMP; MACRO POINTER DURING EXPANSION
INT INM; NUMBER OF MACROS
INT IPR(010); PARAMETER VALUES
INT ISS(040);    INT TO HOLD NUMBERS-MAIN STACK
INT ITU; VALUE OF PARAMETER TO USE
INT IUU; SYMBOL GENERATOR(UNIQUE)
INT IXX; WORK
INT IYY; WORK
BEGINMAIN(AC,IAV)
NL=+KCR
LF=+KLF
GOSUB CR

MS 'COPYRIGHT'
MS ' (C) 1981'
MS ' W.A.GALE'
GOSUB CR
MS '8080 TINC'
MS 'MP COMPIL'
MS 'ER CP/M V'
MS 'ERSION 1.'
MS '05/TINCMP'
GOSUB CR
MS 'COPYRIGHT'
MS ' (C) 1981'
MS ' A L BEND'
MS 'ER, MD   '
GOSUB CR
GOSUB IN
GOSUB RM
LOC 00
WHILE
       READ CC FROM F1
       AA=ER==CO
ON AA; THAT IS, UNTIL EOF IS REACHED ON INPUT

     IF UG

               WHILE
                        BB=CC==NL
```

(Continued on next page)

```
                                DD=CC==LF
                                AA=CC==OG
                                EE=BB?DD
                                BB=CC==BL
                                AA=BB?AA
                                AA=EE?AA
                        ON AA; IGNORE LEADING CHARACTERS
                        GOSUB GC; READ CC FROM F1
                ENDWHILE
ENDIF
BP=C1; BUF POINTER
BF(C0)=CC
WHILE
                GOSUB GC; READ CC FROM F1
                        AA=CC==NL
                        IF AA
                        GOSUB GC; READ CC FROM F1
                        AA=CC==LF
                        IF AA
                        CC=NL
                        ENDIF
                        ENDIF
                AA=CC!=NL
                BB=BP!=C8
                AA=AA&BB
ON AA; WHILE LESS THAN 80 CHAR AND NOT NEWLINE
                BF(BP)=CC; THEN PUT IT IN BUFFER FOR MULT COMP
                BP++

  ENDWHILE
  WHILE
  AA=CC!=NL
  ON AA


                GOSUB GC; READ CC FROM F1
  ENDWHILE; HERE WE ARE DUMPING A LONG INPUT LINE
  BF(BP)=RC
  BP++
  BF(BP)=NL
  LE=BP
  AA=BP<=MM
  IF AA; TOO SHORT TO MATCH
                ML=+000
                GOTO 17
  ELSE
                ML=+001
  ENDIF
  IDP=I00
  PP=C0
  IJJ=I00
  INM=C0
  WHILE
                AA=IDP<!IED; DEF PTR < END OF DEFINITIONS
  ON AA
                BP=C0
                WHILE
                        AA=BP<=LE
                ON AA
                        AA=LS(IJJ)
                        AA=AA==RC
```

```
                O3=BF(BP)
                O3=O3==RC
                AA=AA&O3;  CHECK EOL MATCH TARG & TEMPLATE
                IF AA
                        GOSUB DM;  DO MACRO EXPANSION
                        GOTO OO
                ELSE
                AA=BF(BP)
                BB=LS(IJJ)
                AA=AA==BB
                IF AA
                        GOTO O1;  MATCHING

                        ELSE
                        AA=BB!=SF;  NOT A TEMPLATE PARAMETER FLAG
                        IF AA
                                GOTO 10;  MISMATCHED
                        ELSE;  THIS IS A PARAMETER
                                PP++
                                AA=BF(BP)
                                IAA=AA
                                IPR(PP)=IAA
                        ENDIF
                        ENDIF
                        ENDIF
                LOC O1

                        BP++
                        IJJ++
                ENDWHILE
        LOC 10
                PP=CO
                INM++
                IDP=ILP(INM)
                IJJ=IDP
        ENDWHILE
  LOC 17
        BP=CO
        WHILE
                CC=BF(BP)
                O1=BP+C1
                AA=BF(O1)
                AA=AA!=NL
        ON AA
                IF ML;  THEN ALSO WRITE
                        WRITE CC
                ENDIF
                WRITE CC INTO F2
                BP++
        ENDWHILE
        IF ML
                GOSUB CR
        ENDIF
        IF UN;  ONLY IF NOT SUPPRESSING
                WRITE NL INTO F2
                WRITE LF INTO F2
        ENDIF
  ENDWHILE
LOC 88; END OF SATISFACTORY COMPILATION
  MS 'TINCMP CO'
  MS 'MPILATION'
  MS ' FINISHED'
GOSUB CR;
  CLOSE F1
  CLOSE F2
```

(Continued on next page)

```
 ENDMAIN
                SUB GC; GETS THE NEXT CHARACTER INTO CC GOES TO 88 ON END
        READ CC FROM F1
        AA=ER!=C0
        IF AA; IF NOT NORMAL READ OPERATION
                GOTO 88; !!!! NOT GOOD PROGRAMMING PRACTICE AT ALL !!!!
        ENDIF
        AA=CC==EF; IF CHARACTER WAS EOF MARK
        IF AA; IN CP/M SYSTEM THIS CAN BE RETURNED TO USER
                GOTO 88; !!!! NOT GOOD PROGRAMMING PRACTICE AT ALL !!!!
        ENDIF
                ENDSUB; GC - GET CHARACTER FROM INPUT FILE
                SUB SD;   CONVERTS TOUSE TO A NUMBER WITHOUT ZRO LEADING
AA=ITU<!I00
IF AA
        BB=+001
        ITU=-ITU
ELSE
        BB=+000
ENDIF
AA=ITU==I00
IF AA
        ND=C1
        DS(C0)=ZR
ELSE
        ND=C0
        WHILE
                AA=I00<!ITU
        ON AA
                IYY=ITU/I10
                IAA=I10*IYY
                IXX=ITU-IAA
        ITU=IYY
        AA=IXX

        AA=AA+ZR
        DS(ND)=AA
        ND++
        ENDWHILE
ENDIF
DS(ND)=OR
ND=ND+BB;  INCR FOR NEG INTEGER ONLY
ENDSUB

                SUB WN;   WRITE NUMBER INTO F2

GOSUB SD;  STACK THE DIGITS
WHILE;  NOW WRITE THEM OUT FIRST TO LAST
        IAA=ND
        AA=I00<!IAA
ON AA
        ND--
        AA=DS(ND)
        WRITE AA INTO F2
ENDWHILE
ENDSUB

                SUB PN;  WRITE THE NUMBER ON THE TERMINAL

GOSUB SD;  STACK THE DIGITS
WHILE
```

```
        IAA=ND
        AA=I00<!IAA
ON AA
        ND--
        AA=DS(ND)
        WRITE AA
ENDWHILE
WRITE BL
ENDSUB

                SUB CD;  CONVERT AA AS A DECIMAL DIGIT

BB=ZR<=AA
CC=AA<=C9
BB=BB&CC
IF BB

        AA=AA-ZR
        RETURN

ENDIF
AA=C0
ENDSUB

                SUB CH;  CONVERT AA AS HEX DIGIT

BB=ZR<=AA
CC=AA<=C9
BB=BB&CC
IF BB
        AA=AA-ZR
        RETURN
ENDIF
BB=HA<=AA
CC=AA<=HF
BB=BB&CC
IF BB
        AA=AA-HA
        AA=AA+CX

        RETURN
ENDIF
AA=C0
ENDSUB

                SUB IN;  INITIALIZE

ILM=+08920
I00=+00000
I01=+00001
I10=+00010
I09=+00009
C0=+000
C1=+001
C2=+002
C3=+003
EF=+KEF
C4=+040
C8=+080
I16=+00016
SP=+000
C9='9'
ZR='0'
BL=' '
HF='F'
```

(Continued on next page)

```
HA='A'
CX=+010
IBC=I01
TR='R'
CLOSE F1
ASSOCIATE FCB 1 WITH IBC
OPEN F1 FOR TR AT IBC
TR='W'
IBC++
CLOSE F2
ASSOCIATE FCB 2 WITH IBC
OPEN F2 FOR TR AT IBC
READ AA FROM F1; X SUPPRESSES NEW LINE OUTPUT
OT='T'
UT=+000
BB='X'
UN=AA!=BB; UN SAYS CHARACTER WAS NOT X SO DONT SUPPRESS
READ RB FROM F1
READ RC FROM F1; COMMENT AND EOL FLAG
READ SF FROM F1; TEMPLATE PARAMETER FLAG
READ MF FROM F1; EXPANSION OPERATION FLAG


BB='X'
AA=MF==BB
IF AA
        UO=C0
ELSE
        UO=C1
ENDIF
OP='P'; PARAMETER DESIGNATOR IN OPERATION SEQUENCE
OE='@'; ESCAPE CHARACTER
OD='V'; CONVERT PARAMETER TO DIGIT IN ACTION SEQUENCE
OB='!'; POP STACK DESIGNATOR IN OPERATION SEQUENCE
OS='S'; STACK DESIGNATOR IN OPERATION SEQUENCE
OH='H'; HEX CONSTANT FETCH AND WRITE

ON='N'; LITERAL NUMERIC FETCH
OL='L'; LITERAL BYTE FETCH
OC='C'; CHARACTER OUT DESIGNATION
OA='+'; ADD TO STACK DESIGNATION
OR='-'; SUBTRACT (REDUCE) FROM STACK
OM='*'; MULTIPLY STACK BY BASE AND ADD
READ OG FROM F1; IGNORE CHARACTER
AA='X'
BB=AA==OG
IF BB
        UG=+000
ELSE
        UG=+001
ENDIF
READ CC FROM F1; NEW LINE
AA=NL!=CC; NL IS NEWLINE
IF AA
        MS 'FLAG LINE'
        STOP 1
ENDIF
IUU=+00100
ENDSUB; IN
                SUB RM;  READ MACROS
III=I00
```

```
INM=C0
MM=+127
WHILE
        READ CC FROM F1
        AA=ER==C0

ON AA
        CHOOSE ON CC
        CASE OE;ACCEPT THE NEXT CHARACTER UNCRITICALLY
                READ CC FROM F1
                GOTO 77
        CASE RB;BEGIN A DEFINITION
                ILP(INM)=III
                INM++
                ML=+000

        CASE NL;IGNORE
        CASE LF;IGNORE
        CASE RC;IGNORE FOLLOWING COMMENTS AND MARK LINE END
                LS(III)=RC
                III++
                AA=ML<!MM
                IF AA;THIS LINE IS SHORTEST YET
                        MM=ML

                ENDIF
                WHILE
                        READ CC FROM F1


                        AA=CC!=LF
                ON AA
                ENDWHILE
        CASE OG;IF USING IGNORE, IGNORE
                IF UG
                ELSE
                        GOTO 77


                ENDIF
        DEFAULT;
        LOC 77
                LS(III)=CC
                III++
                AA=ILM<!III
                IF AA
                        MS 'MACMEMXST'
                        GOSUB CR
                        CLOSE F1
                        STOP 5
                ENDIF
                ML++
        ENDCHOOSE
ENDWHILE
AA=CC!=EF
IF AA
        MS 'DEFN READ'
        STOP 2
ENDIF
CLOSE F1
IBC=+00003
ASSOCIATE FCB 3 WITH IBC
TR='R'
OPEN F1 FOR TR AT IBC
IED=III;END OF DEFINITIONS
```

(Continued on next page)

```
MS 'LOADED...'
ITU=III
GOSUB PN
MS '.BYTES FO'
MS 'R DEFINES'
GOSUB CR
ILP(INM)=III
ITU=INM
GOSUB PN
MS '.MACROS..'
ITU=MM
GOSUB PN
MS ' MIN LEN.'
GOSUB CR
ENDSUB; RM
                SUB CR; DO CARRIAGE RETURN/LINE FEED SEQUENCE
WRITE NL
WRITE LF
ENDSUB; CR

                SUB DM; DO MACRO EXPANSION

IMP=IJJ+IO1
INM++
IDP=ILP(INM)
WHILE
        AA=IMP<!IDP
ON AA; UNTIL WE HAVE READ UP TO THE NEXT MACRO DEFINITION
        AA=LS(IMP)
        IF UO
                AA=AA==MF


ELSE
         AA=CO
ENDIF
IF AA; OPERATION CODE
         IMP++
         O1=LS(IMP); FROM INDICATOR
         IMP++
         AA=LS(IMP)
         O2=AA
         GOSUB CD; FOR DIGIT CONVERSION
         DG=AA
         IMP++
         O3=LS(IMP); DESTAD
IF UT
                  WRITE O1
                  WRITE O2
                  WRITE O3
ENDIF
         CHOOSE ON O1
         CASE OP; FETCH PARAMETER
         ITU=IPR(DG)
         CASE OD; CONVERT FROM DIGIT TO CHARACTER
         IAA=IPR(DG)
         AA=IAA
         GOSUB CD
         ITU=AA
         CASE OB; POP STACK
                  ITU=ISS(SP)
```

```
        AA=SP<=C0
        IF AA
                MS 'S STACKER'
                GOSUB CR
                SP=C1
        ENDIF
        SP--

CASE OS; FETCH FROM STACK WITHOUT POPPING IT
        ITU=ISS(SP)
CASE OH; FETCH AND WRITE HEX CONSTANT BYTE
        AA=02
        GOSUB CH
        IAA=AA
        IAA=IAA*I16
        AA=03
        GOSUB CH
        IBB=AA
        ITU=IAA+IBB
        03=0C

CASE OL; LITERAL BYTE FETCH
        ITU=02
CASE ON; LITERAL DIGIT FETCH
        AA=02
        GOSUB CD
        ITU=AA
CASE OT; TURN ON TRACE MODE
        UT=+001
DEFAULT; FETCH  A UNIQUE NUMBER
        ITU=IUU
        IUU++

ENDCHOOSE
IF UT
        III=ITU
        GOSUB PN
        ITU=ISS(SP)
        GOSUB PN
        ITU=SP
        GOSUB PN



                ITU=III
                GOSUB CR
        ENDIF
        CHOOSE ON 03
        CASE 0C; CHARACTER OUTPUT
                AA=ITU
                WRITE AA INTO F2
        CASE OS; PUT ON STACK
                SP++
                AA=C4<=SP
                IF AA
                        MS 'S OVERFLO'
                        GOSUB CR
                        SP=C4
                ENDIF
                ISS(SP)=ITU
```

(Continued on next page)

```
                CASE OP; PUT INTO PARAMETER LOCATION
                        IPR(DG)=ITU
                CASE OA; ADD TO STACK
                        IAA=ISS(SP)
                        IAA=IAA+ITU
                        ISS(SP)=IAA
                CASE OR; REDUCE (SUBTRACT) FROM STACK
                        IAA=ISS(SP)
                        IAA=IAA-ITU
                        ISS(SP)=IAA
                CASE OM; MULTIPLY BY BASE AND ADD
                        IAA=ISS(SP)
                        IAA=IAA*I10
                        IAA=IAA+ITU
                        ISS(SP)=IAA
                CASE OH; OUTPUT HIGH BYTE
                        UNPACK(ITU,AA,BB)
                        WRITE AA INTO F2
                DEFAULT; WRITE OUT AS A DECIMAL NUMBER
                        GOSUB WN
                ENDCHOOSE
        ELSE; END OF ACTION SECTION
        AA=LS(IMP)
        IF UN
                BB=AA!=RC
        ELSE
                BB=C1
        ENDIF
        IF BB
                WRITE AA INTO F2
        ELSE
                WRITE NL INTO F2
                WRITE LF INTO F2
        ENDIF
        ENDIF
        IMP++
ENDWHILE
UT=+000
ENDSUB; DM
BOTTOM; END OF TINCMP 8080 CP/M COMPILER
```

End Listing I

Listing II

```
U:;$^
:TOP;
        @       EXTRN@  ?MSG;
        @       EXTRN@  ?BOOT,?INIT,?ENDING;
        @       COMMON@ /IOCTRL/;
        .ER@:@  DB@     0;
        .AC@:@  DB@     0;
        .IAV@:@ DW@     0,0,0,0,0,0;
        @       COMMON@ /       / ;
:BEGINMAIN(AC,IAV);
        @       DSEG;
        .ERASER.@: DB@  0;
        @       CSEG;
?SETAC@:@       MVI@    A,1;
        @       RM;
        @       RZ;
        @       XRA@    A;
        @       RET;
```

```
?INTCK@:@       MVI@    B,1;
        @       LDAX@   D;
        @       CMP@    M;
        @       JNZ@    @$+8;
        @       INX@    H;
        @       INX@    D;
        @       LDAX@   D;
        @       XRA@    M;
        @       RZ;
        @       MOV@    A,B;
        @       RET;

?START@:@       LXI@    SP,?STACK;
@       CALL@   ?INIT;
:PARAMETER K$$=$$$;
:MS '$$$$$$$$$$';
@       CALL@   ?MSG;
@       DB@     '^P1C^P2C^P3C^P4C^P5C^P6C^P7C^P8C^P9C';
:GOSUB $$;
        @       CALL@   ..^P1C^P2C;
:GOTO $$;
        @       JMP@    ?LL0^P1C^P2C;
:$$=+000;
        @       XRA@    A;
        @       STA@    .^P1C^P2C;
:I$$=+00000;
        @       LXI@    H,0;
        @       SHLD@   .I^P1C^P2C;
:I$$=+K$$;
        @       LXI@    H,.K^P3C^P4C;
        @       SHLD@   .I^P1C^P2C;
:PACK(I$$,$$,$$);
        @       LDA@    .^P5C^P6C;
        @       MOV@    L,A;
        @       LDA@    .^P3C^P4C;
        @       MOV@    H,A;
        @       SHLD@   .I^P1C^P2C;



:UNPACK(I$$,$$,$$);
        @       LHLD@   .I^P1C^P2C;
        @       MOV@    A,L;
        @       STA@    .^P3C^P4C;
        @       MOV@    A,H;
        @       STA@    .^P5C^P6C;
:CALL I$$;
        @       LXI@    H,@$+8;
        @       PUSH@   H;
        @       LHLD@   .I^P1C^P2C;
        @       PCHL;
:$$=+K$$;
        @       MVI@    A,.K^P3C^P4C;
        @       STA@    .^P1C^P2C;
:$$='$';
        @       MVI@    A,'^P3C';
        @       STA@    .^P1C^P2C;
:$$=+$$$;
        @       MVI@    A,^V3S^V4*^V5*^!0N;
        @       STA@    .^P1C^P2C;
:I$$=+$$$$$;
        @       LXI@    H,^V3S^V4*^V5*^V6*^V7*^!0N;
        @       SHLD@   .I^P1C^P2C;
:$$=$$;
        @       LDA@    .^P3C^P4C;
        @       STA@    .^P1C^P2C;
:$$++;
        @       LXI@    H,.^P1C^P2C;
        @       INR@    M;
:$$--;
        @       LXI@    H,.^P1C^P2C;
        @       DCR@    M;
:I$$=$$;
        @       LHLD@   .^P3C^P4C;
        @       MVI@    H,0;
        @       SHLD@   .I^P1C^P2C;
:$$=I$$;
        @       LDA@    .I^P3C^P4C;
        @       STA@    .^P1C^P2C;
:I$$=I$$;
        @       LHLD@   .I^P3C^P4C;
        @       SHLD@   .I^P1C^P2C;
:$$=$$($$);
        @       LHLD@   .^P5C^P6C;
        @       MVI@    H,0;
        @       LXI@    D,.^P3C^P4C;
        @       DAD@    D;
        @       MOV@    A,M;
        @       STA@    .^P1C^P2C;
:I$$=I$$($$);
        @       LHLD@   .^P5C^P6C;
        @       MVI@    H,0;
        @       LXI@    D,.I^P3C^P4C;
        @       DAD@    H;
        @       DAD@    D;

        @       LXI@    D,.I^P1C^P2C;
        @       MOV@    A,M;
        @       STAX@   D;
        @       INX@    D;
        @       INX@    H;
        @       MOV@    A,M;
        @       STAX@   D;
:$$=$$(I$$);
        @       LHLD@   .I^P5C^P6C;
        @       LXI@    D,.^P3C^P4C;
        @       DAD@    D;
        @       MOV@    A,M;
        @       STA@    .^P1C^P2C;
:I$$=I$$(I$$);
        @       LHLD@   .I^P5C^P6C;
        @       LXI@    D,.I^P3C^P4C;
        @       DAD@    H;
        @       DAD@    D;
        @       MOV@    E,M;
        @       INX@    H;
        @       MOV@    D,M;
        @       XCHG;
        @       SHLD@   .I^P1C^P2C;
:$$($$)=$$;
        @       LHLD@   .^P3C^P4C;
        @       MVI@    H,0;
        @       LXI@    D,.^P1C^P2C;
        @       DAD@    D;
        @       LDA@    .^P5C^P6C;
        @       MOV@    M,A;
:I$$($$)=I$$;
        @       LHLD@   .^P3C^P4C;
        @       MVI@    H,0;
        @       DAD@    H;
        @       LXI@    D,.I^P1C^P2C;
        @       DAD@    D;
        @       LXI@    D,.I^P5C^P6C;
        @       LDAX@   D;
        @       MOV@    M,A;
        @       INX@    D;
        @       INX@    H;
        @       LDAX@   D;
        @       MOV@    M,A;
:$$(I$$)=$$;
        @       LHLD@   .I^P3C^P4C;
        @       LXI@    D,.^P1C^P2C;
        @       DAD@    D;
        @       LDA@    .^P5C^P6C;
        @       MOV@    M,A;
:I$$(I$$)=I$$;
        @       LHLD@   .I^P3C^P4C;
        @       LXI@    D,.I^P1C^P2C;
        @       DAD@    H;
        @       DAD@    D;
        @       XCHG;


        @       LHLD@   .I^P5C^P6C;
        @       MOV@    A,L;
        @       STAX@   D;
        @       INX@    D;
        @       MOV@    A,H;
        @       STAX@   D;
:$$=$$+$$;
        @       LXI@    H,.^P5C^P6C
        @       LDA@    .^P3C^P4C;
        @       ADD@    M;
        @       STA@    .^P1C^P2C;
:$$=$$-$$;
        @       LXI@    H,.^P5C^P6C
        @       LDA@    .^P3C^P4C;
        @       SUB@    M;
        @       STA@    .^P1C^P2C;
:$$=$$&$$;
        @       LXI@    H,.^P5C^P6C
        @       LDA@    .^P3C^P4C;
        @       ANA@    M;
        @       STA@    .^P1C^P2C;
:$$=$$?$$;
        @       LXI@    H,.^P5C^P6C
        @       LDA@    .^P3C^P4C;
        @       ORA@    M;
        @       STA@    .^P1C^P2C;
```

(Continued on next page)

```
:I$$++;
        @       LHLD@   .I^P1C^P2C;
        @       INX@    H;
        @       SHLD@   .I^P1C^P2C;
:I$$--;
        @       LHLD@   .I^P1C^P2C;
        @       DCX@    H;
        @       SHLD@   .I^P1C^P2C;
:I$$=-I$$;
        @       LHLD@   .I^P3C^P4C;
        @       CALL@   ?ICOMP;
        @       EXTRN@  ?ICOMP;
        @       SHLD@   .I^P1C^P2C;
:I$$=I$$+I$$;
        @       LHLD@   .I^P5C^P6C;
        @       XCHG;
        @       LHLD@   .I^P3C^P4C;
        @       DAD@    D;
        @       SHLD@   .I^P1C^P2C;
:I$$=I$$-I$$;
        @       LHLD@   .I^P5C^P6C;
        @       XCHG;
        @       LHLD@   .I^P3C^P4C;
        @       CALL@   ?ISUB;
        @       EXTRN@  ?ISUB;
        @       SHLD@   .I^P1C^P2C;
:I$$=I$$*I$$;
        @       LHLD@   .I^P5C^P6C;
        @       XCHG;


        @       LHLD@   .I^P3C^P4C;
        @       CALL@   ?IMUL;
        @       EXTRN@  ?IMUL;
        @       SHLD@   .I^P1C^P2C;
:I$$=I$$/I$$;
        @       LHLD@   .I^P5C^P6C;
        @       XCHG;
        @       LHLD@   .I^P3C^P4C;
        @       CALL@   ?IDIV;
        @       EXTRN@  ?IDIV;
        @       SHLD@   .I^P1C^P2C;
:$$=$$==$$;
        @       LXI@    H,.^P3C^P4C;
        @       LDA@    .^P5C^P6C;
        @       CMP@    M;
        @       JZ@     @$+5;
        @       XRA@    A;
        @       LXI@    B,013EH;
        @       STA@    .^P1C^P2C;
:$$=$$!=$$;
        @       LXI@    H,.^P5C^P6C;
        @       LDA@    .^P3C^P4C;
        @       CMP@    M;
        @       JNZ@    @$+5;
        @       XRA@    A;
        @       LXI@    B,013EH;
        @       STA@    .^P1C^P2C;
:$$=$$<=$$;
        @       LXI@    H,.^P5C^P6C;
        @       LDA@    .^P3C^P4C;
        @       CMP@    M;
        @       CALL@   ?SETAC;
        @       STA@    .^P1C^P2C;
:$$=$$<!$$;
        @       LDA@    .^P3C^P4C;
        @       LXI@    H,.^P5C^P6C;
        @       CMP@    M;
        @       RAL;
        @       ANI@    1;
        @       STA@    .^P1C^P2C;
:$$=!$$;
        @       LDA@    .P3C^P4C;
        @       CMA;
        @       ANI@    1;
        @       STA@    .^P1C^P2C;
:$$=I$$==I$$;
        @       LXI@    D,.I^P5C^P6C;
        @       LHLD@   .I^P3C^P4C;
        @       LDAX@   D;
        @       XRA@    L;
        @       JNZ@    @$+12;
        @       INX@    D;
        @       LDAX@   D;
        @       XRA@    H;
        @       JNZ@    @$+6;


        @       MVI@    A,1;
        @       LXI@    B,003EH;
        @       STA@    .^P1C^P2C;
:$$=I$$!=I$$;
        @       LXI@    D,.I^P5C^P6C;
        @       LXI@    H,.I^P3C^P4C;
        @       CALL@   ?INTCK;
        @       STA@    .^P1C^P2C;
:$$=I$$<=I$$;
        @       LHLD@   .I^P5C^P6C;
        @       XCHG;
        @       LHLD@   .I^P3C^P4C;
        @       CALL@   ?CDEHL;
        @       EXTRN@  ?CDEHL;
        @       CALL@   ?SETAC;
        @       STA@    .^P1C^P2C;
:$$=I$$<!I$$;
        @       LHLD@   .I^P5C^P6C;
        @       XCHG;
        @       LHLD@   .I^P3C^P4C;
        @       CALL@   ?CDEHL;
        @       EXTRN@  ?CDEHL;
        @       RAL;
        @       ANI@    1;
        @       STA@    .^P1C^P2C;
:CASE $$;
        ^!OP^!9P^!8P
        @       JMP@    .XC^P0N;
        @       ORG@    @$-^P9N;
        .UP^P8N@:@      LDA@    .^P1C^P2C;
        @       CMP@    M;
        @       JNZ@    .UP^U1S^S8N^N0S^P0S;
:ASSOCIATE FCB $ WITH I$$;
        @       LXI@    D,?IFCB+((^P1C-1)*36);
        @       LXI@    H,.I^P2C^P3C;
        @       CALL@   ?ATACH;
        @       EXTRN@  ?ATACH,?IFCB;
:WRITE $$;
        @       LDA@    .^P1C^P2C;
        @       CALL@   ?WRNXT;
        @       EXTRN@  ?WRNXT;
:CLOSE $$;
        @       LXI@    H,.^P1C^P2C;
        @       CALL@   ?BCLOF;
        @       EXTRN@  ?BCLOF;
:STOP $;
        @       MVI@    A,^P1C;
        @       CALL@   ?ISTOP;
        @       EXTRN@  ?ISTOP;
:OPEN $$ FOR $$ AT I$$;
        @       LXI@    D,.^P1C^P2C;
        @       LDA@    .^P3C^P4C;
        @       LHLD@   .I^P5C^P6C;
        @       CALL@   ?IOPEN;
        @       EXTRN@  ?IOPEN;


:ENDSUB;
        @       RET;
:RETURN;
        @       RET;
:EXTERNAL PROCEDURE $$;
        @       CALL@   ..$$;
        @       EXTRN@  ..$$;
:READ $$ FROM $$;
        @       LXI@    H,.^P1C^P2C;
        @       LXI@    D,.^P3C^P4C;
        @       CALL@   ?BREAD;
        @       EXTRN@  ?BREAD;
:WRITE $$ INTO $$;
        @       LDA@    .^P1C^P2C;
        @       LXI@    D,.^P3C^P4C;
        @       CALL@   ?BWRIT;
        @       EXTRN@  ?BWRIT;
:ENDMAIN;
        @       CALL@   ?ENDING;
        @       JMP@    ?BOOT;
:LOC $$;
        ?LLO^P1C^P2C@:;
:CHOOSE ON $$;
        @       LDA@    .^P1C^P2C;
```

```
        @       LXI@    H,.ERASER.;
        @       MOV@    M,A^U0S^N3S^U1S;
:DEFAULT;
        ^!0P^!9P^!8P
        @       JMP@    .XC^P0N;
        @       ORG@    $-^P9N;
        .UP^P8N@:;
        ^P0S
:ENDCHOOSE;
        .XC^!0N@:;
:WHILE;
        ^U0S.X^S0N@:;
:ON $$;
        @       LDA@    .^P1C^P2C;
        @       ORA@    A;
        @       JZ@     .W^S0N;
:ENDWHILE;
        @       JMP@    .X^S0N;
        .W^!0N@:;
:SUB $$;
        ..^P1C^P2C@:;
:IF $$;
        @       LDA@    .^P1C^P2C;
        @       ORA@    A;

        @       JZ@     .F^U0S^S0N;
:ELSE;
        @       JMP@    .F^U0P^P0N^!9P;
        .F^P9N@:;
        ^P0S
:ENDIF;

        .F^!0N@:;
:BYTE $$;
        .^P1C^P2C@:@    DS@     1;
:INT I$$;
        .I^P1C^P2C@:@   DS@     2;
:BYTE $$($$$$$);
        .^P1C^P2C@:@    DS@     ^V3S^V4*^V5*^V6*^V7*^!0N;
:BYTE $$($$$);
        .^P1C^P2C@:@    DS@     ^V3S^V4*^V5*^!0N;
:INT I$$($$$$$);
        .I^P1C^P2C@:@   DS@     2*^V3S^V4*^V5*^V6*^V7*^!0N;
:INT I$$($$$);
        .I^P1C^P2C@:@   DS@     2*^V3S^V4*^V5*^!0N;
:BOTTOM;
        @       DSEG;
        @       DS@     60;
        ?STACK@:@       DW@     0;
        @       END@    ?START;
:READBUF $$;
        @       LXI@    D,.^P1C^P2C;
        @       CALL@   ?RDBUF;
        @       EXTRN@  ?RDBUF;
:WRITEBUF $$;
        @       LXI@    D,.^P1C^P2C;
        @       CALL@   ?WRBUF;
        @       EXTRN@  ?WRBUF;
:READ $$;
        @       CALL@   ?RNX;
        @       EXTRN@  ?RNX;
        @       STA@    .^P1C^P2C;
:LOCAL VARIABLES;
        @       DSEG;
:ENDLOCAL;
        @       CSEG;
:EXTERNAL PROCEDURE NAME $$;
        @       PUBLIC@ ..^P1C^P2C;
        @       CSEG;
        ..^P1C^P2C@:;
:PROCEDURE END;
        @       RET;
        @       END;
:;
:REGISTER=$$$;
:LOMEM=$$$$$;
:HIMEM=$$$$$;
```

End Listing II

BOOK REVIEWS

The 8086 Book (Includes the 8088)
By **Russell Rector and George Alexy**
Published by **Osborne/McGraw-Hill**
$16.99, about 550 pages
Reviewed by **Ray Duncan**

Publishers have taken notice of the rapidly expanding, highly profitable microprocessor marketplace and have begun to flood computer stores with handbooks. All too often, these are poorly written and indexed, shot through with inaccuracies, or just blatant ripoffs created by paraphrasing the manufacturer's technical literature.

Happily, there are exceptions. *The 8086 Book* is, for the most part, a lucid, well-organized, detailed guide to the Intel 16-bit microprocessor. Every programmer working with the 8086 or 8088 devices will find this book helpful.

Before describing the good features of this book, I would like to do a little editorializing of my own. It seems that most of the handbooks traditionally begin with few pages giving a bonehead explanation of "what is a computer" on approximately the fourth-grade level, just before diving into the technical details directed at skilled engineers and programmers with years of computer experience. This book follows the trend, and it is hard to understand how the authors could write some of the material with a straight face. A quote from chapter one:

"The CPU operates with 1s and 0s."

Could that statement possibly be aimed at the same audience which is expected to comprehend, in chapter 8, "The first up-down counter tracks the depth of the queue, while a second up-down counter captures the queue depth on a match"? Please, budding authors, don't try to make your books be all things to all people.

After disposing of the foundations of computer science in about thirty pages, we get to a beautifully laid-out section on the 8086 instruction set, which comprises the majority of the book. Mnemonics are explained, one to a page in alphabetical order. Since the 8086 has a large number of addressing modes which are sometimes hard to visualize, many specific examples are supplied together with their effects on the various CPU registers and flags. Diagrams are used liberally.

There follows a set of tables of instructions grouped by their various classes; i.e., data movement, arithmetic, shifts, transfers, etc. This is handy for those occasions when you know what you need the computer to do but can't remember the mnemonic name, or aren't sure whether an addressing mode is available.

The last few short chapters contain technical information concerning the 8086 and 8088 microprocessors, such as timing diagrams and specifications of bus signals; largely a rehash of the Intel literature which is available for free. Most of this would be unintelligible to a programmer without an engineering background.

The 8086 Book will be most useful to experienced assembly language programmers with a fair amount of hardware expertise. It is not a guide to programming style on the 8086, although a few illustrative programs are sprinkled throughout the text. As a reference tool, it is well worth the money and is far superior to any other book about the 8086 I have seen.

Pascal Programming Structures: An Introduction to Systematic Programming
By **George W. Cherry**
Published by **Reston Publishing Co.**
$14.95, 313 pages
Reviewed by **Richard L. Lozes**

The first thing I noticed about this book was that it mentions no institutional affiliation of the author. Nonetheless, the overall structure and particular details lead me to believe that the author has had no meager experience in teaching.

Pascal Programming Structures is a well-conceived text, meeting the needs of the novice programming student. It should also serve well, as it has in my case, to initiate hackers of other languages, to the entirety of this now-famous language, Pascal. (This last statement begs clarification – author Cherry claims completeness, but without the ANSI standard at hand I had no way to verify that.)

A student should never feel that he needs to read the next chapter before being able to write real programs. Mr. Cherry is quite emphatic about displaying full, albeit simple, programs from the start. The text increases in difficulty as it moves along, making it simple for the professor trying to squeeze the last bit of information into the end of a course.

Two positive remarks are in order. First, the traditional fog of bits, words, machine states, numerical representations, and other points ordinarily found in introductory programming texts, but wholly lacking relevance, are absent. Second, the useful techniques of programming (top-down development, stepwise refinement) are presented and illustrated without the preachiness which has marred many other texts of recent vintage.

Two negative comments are also in order. I found the style a bit too relaxed; I prefer not to see subject-verb contractions, although I suppose the TV generation will not notice. Far more important, I dislike intensely the author's practice of bracketing remarks by the likes of "Remark." . . . "End of remark." The English language already has conventions to serve this purpose: the parenthesis pair, and the footnote. The author seems to have confused English with Pascal, and in the interest of avoiding C. P. Snow's "two cultures," that confusion should never have reached print.

Nevertheless, a solid introductory programming course could be built around this text. With determination, the entire text could be covered in a semester to good effect.

Build Your Own Z80 Computer
By **Steve Ciarcia**
Published by **BYTE Books/McGraw-Hill**
$15.95, 332 pages
Reviewed by **George W. Jolly**

Have you ever thought that only an electrical engineer could design his own computer? I had begun to think so when this book came along. If you have a basic understanding of electronics, and a desire to build (or understand) your own microcomputer, then this book is for you. It is for technically-minded people who really want to learn. It assumes the ability to read schematic diagrams, and knowledge of common electronic components. It is a valuable book even if you do not choose to build a computer.

Although the subject is a Z80-based computer, the text has wider application. Ciarcia presents the computer's capabilities as a series of building blocks that you may arrange as you need. Most of these capabilities are not dependent on the choice of microprocessor. Design decisions are discussed in a straightforward manner, without introducing purely theoretical problems. The book's emphasis is on designing with standard, off-the-shelf components. *Build Your Own Z80*

Computer offers fascinating insights into the computer design process.

As the basic system circuits are presented, procedures for testing them are explained. These discussions show you how computers work internally. The subject of testing input and output introduces the concept of diagnostic software. Tiny programs are presented that allow you to test most of the basic computer, heightening your appreciation of the relationships between the parts of a computer.

The system described is a single-board computer intended for a beginner's experimentation. A monitor program is given, complete with assembled listing. Even better, flow charts for the functional modules of the monitor are shown and discussed at some length. The monitor handles a hexadecimal keypad similar to those found on commercially available single-board evaluation systems. Commands are provided to examine and change memory or registers, and to execute a program in memory. The display consists of six output ports which drive hexadecimal display chips.

Other peripherals covered are: a 2708 EPROM programmer, an RS-232C serial port, a cassette-tape storage interface, and analog/digital conversions. For the more ambitious builder, an entire chapter is devoted to the design of a CRT display. The circuit shown is a 16-line, 64-character format, but notes on design of a 24-by-80 display are also provided. Two integrated circuits do most of the work in this circuit. Most of the design work demanded by the project involves choosing parameter values to feed into these chips. The CRT display is not a memory-mapped video board, but a separate terminal using a serial port. A schematic for the serial port and ASCII keyboard encoder is provided, but not discussed at length.

One particularly pleasant surprise in the book is the long tutorial on analog/digital conversion. Several methods are explained, complete with actual circuits. One of the circuits is taken from a digital voltmeter; it differs from other circuits in that its results are in binary-coded decimal digits. A driver program is shown that can operate this circuit. Equally important, the basic theory of signal sampling is presented. Such possible applications as voice reproduction and recording temperatures are discussed.

Build Your Own Z80 Computer provides appropriate technical information about the less-common integrated circuits used. Data sheets for many of the integrated circuits used in the book, and the instruction set descriptions for the Z80 processor are provided.

If you do wish to build from this book, you should read the construction techniques in appendix A. Remember, you will be prototyping your own layout, not "painting by the numbers." Layout considerations are pointed out, but you will need to know how to physically build circuits. Appendix A also lists a company that can provide pre-programmed EPROMS and kits.

Ciarcia's style is friendly and conversational. His pleasant manner makes for easy reading, an invaluable asset. His confidence and enthusiasm will inspire you. *Build Your Own Z80 Computer* is not a handholding construction book for those without electronics knowledge. However, this book offers a readable sampling of the author's considerable experience.

DDJ

SOFTWARE REVIEW

The WORD
A proofreading, word look-up and writing toolkit for users of word processors.

8" single density CP/M diskette (North Star, Superbrain, Apple w/Softcard); includes superb manual with clear instructions and examples.

$75 from

Oasis Systems
2765 Renard Way
San Diego, CA 92103

Reviewed by Gene Head

OASIS SYSTEMS in San Diego did three really remarkable things when they developed and began marketing The WORD, a spelling checker for CP/M files.

First, they managed to cram a 40,000+ word dictionary on a mini-floppy disk and still have room for the operating programs!

Second, the entire spelling package costs just $75.00.

Third, and especially important in my mind, is the fact that they have maintained the concept of software "tools" and have avoided the locked-in, do-everything, menu-driven mess that seems to be increasing in popularity among programmers!

Overview

The WORD is a package of software tools designed to eliminate spelling errors in files created in the CP/M environment. Usually these files would be created by a word processor, but The WORD will work with any ASCII file. My version was delivered on a single North Star mini-floppy with "un-packing" instructions. To "un-pack" the disk, two operational disks are copied from the master disk. Each disk has a copy of the main dictionary and copies of programs to meet specific needs.

The manual is forty-two pages long and very comprehensive. Examples are used to walk the reader through the details of every tool. Operation is quite simple even though a complete proofreading session requires several steps. Here is where the "tools" concept has won out over the single-key-stroke wonder. Each program is independent of the others and each performs a unique function. Generally, the procedure to proofread a text file requires the execution of the SPELL, REVIEW and the MARK programs. Here is how it flows:

SPELL

The SPELL program will check every unique word in the file against the words in the main dictionary and any special dictionary the operator designates. Options include "switches" to skip words with all capital letters (like XYZ INC.), or lines that begin with a period or colon. Many word processors use the period or colon in the first line position to indicate dynamic output formatting, especially .C+ for centering or .L63 for maximum line length.

Un-matched words, words not located in the dictionaries, can be listed, printed, and/or written to a file called ERRWORDS.TXT. In the normal three-step operation, the file ERRWORDS.TXT is used by both the REVIEW and MARK programs.

REVIEW

REVIEW looks at the ERRWORDS.TXT file. The ERRWORDS.TXT file contains all the words *not* found in the dictionaries. Now, it is obvious that not every word in your text file will be found in the dictionaries. Technical words, especially, may have been omitted. The REVIEW program will automatically display each un-matched word from the ERRWORDS.TXT file and request operator action. The operator has four options with each word.

Add word to:
U>pdate dictionary
S>PECIALS.CMP file
M>isspelled words list
D>iscard

Other options:
P>revious word
N>ext word
R>esume review
Q>uit and return to CP/M

Note: The main dictionary can not be altered by the operator.

The UPDATE dictionary is an extension of the main dictionary and will automatically be searched whenever it is present on a disk with the main dictionary in the UPDATE file. Less common words of a special nature (like jargon or acronyms) would go into the SPECIAL dictionary. Unlike the UPDATE dictionary file, the SPECIAL dictionary file is searched *only* if the special search switch is activated. Of course, some words (like personal names) are so unique they should just be discarded.

The other options allow for stepping forward or backward through the list. Additionally, after a word has been processed and recalled by the Previous word option, the current status of the word is displayed instead of the hyphen.

—> ALPHA D

This prompt indicates the word ALPHA has been previously set to be Discarded.

Finally, REVIEW will write the appropriate words to the UPDATE, SPECIAL, and ERRWORDS files. The MARK program will now use the ERRWORDS file that has been REVIEWED and contains only words that are really misspelled.

MARK

MARK is the simplest program and has only one option. When executing MARK a file must be specified as the file to be marked.

A>MARK B:TEXTFILE.PCL

The above example will cause the file named TEXTFILE.PCL on drive B to be searched for matches to the words in the ERRWORDS.TXT file. Whenever a match is found the misspelled word is marked with a "*". The original TEXTFILE.PCL is renamed TEXTFILE.BAK and a new file named TEXTFILE.PCL is written and includes the inserted "*" to denote misspelled words. The word processor can now be used to locate "*" and all misspelled words. Editing commands of the word processor should make fixing the misspelled words especially easy. I mentioned earlier the use of only one option when running MARK. If you want to use a different character to mark the misspelled words, the option is:

A>MARK B:TEXTFILE.PCL $M#

The "$M" sets the mark character switch and the new mark character becomes the "#" character.

Additional Programs

Two similar programs, LOOKUP and FIND, will help you locate the correct spelling if the word is in any of the dictionaries. The FIND program searches for words and recognizes both the "?" and the "*" as wild-card characters. If, for instance, you want to spell ABASEMENT but can't remember if the word is spelled ABASMENT or ABASEMENT then the

following command would return all the words beginning with ABAS regardless of actual total length.

A>FIND ABAS*

Likewise, to locate all words in the dictionary that are exactly five letters long and begin with the letters "SHO" use the following command line.

A>FIND SHO??

The dictionaries have an internal index table for each letter of the alphabet. The search algorithms utilize this index table to decrease search times. If the first letter of the word in question is given, only the words beginning with that letter will be searched. In the last example only the words beginning with the letter "S" will be searched. Avoid especially a wildcard "?" in the first character position as this will cause the entire dictionary to be searched!

LOOKUP uses a different approach based on the assumption that most misspelled words fall into one of four categories:

1. Added extra letters (ACCUTE instead of ACUTE)
2. Dropped a needed letter (ABSESS instead of ABSCESS)
3. One letter wrong (ACCUR instead of OCCUR)
4. Two letters reversed (ACER instead of ACRE)

Using these four assumptions the proper spelling of most misspelled words can be found in the dictionary file.

For those who still have trouble with THERE and THEIR or CAPITOL and CAPITAL, the HOMONYM HELP program will locate most common homonyms and mark them in the text file. Obviously, this helper can't choose which homonym is the correct usage but it can quickly locate possible trouble spots for careful consideration.

In addition to the three dictionaries already mentioned (MAIN, UPDATE and SPECIAL) the user can merge any dictionary (except the MAIN dictionary) with any other dictionary using the PIP program in CP/M. Several SPECIAL and UPDATE dictionaries can be combined and then sorted using the DICTIONARY SORT program. All combined dictionaries *must* be sorted before they can be properly used by the SPELL program. Remember, the dictionaries have an index to speed up searches.

Finally, the WORD COUNT program and the WORD FREQUENCY program are especially useful to writers. The first simply counts the words in the text file. The second will determine the frequency of each word. The output can be to a text file called TEXTFILE.FRQ or to the printer or the console. The output can be sorted either alphabetically or by frequency of usage. Sorted by frequency of usage, the most used word (usually "the") will be listed first with the number of times it was used in the text. Getting a little tired of seeing the word "especially" in this article? It was used seven times, or about five times too many for easy reading.

Conclusion

For those seeking a single-key-stroke approach, this can almost be achieved using CP/M's SUBMIT program with the proper instruction file (.SUB).

All of the programs are relatively slow on my 2 MHz, mini-floppy system. However, the manual indicates significant speed increase for CPU's running at 4 MHz or better, and eight-inch floppies or hard disks. It took about four minutes and ten seconds to completely correct this article with ten words originally misspelled. The correct spelling of every misspelled word was located using the LOOKUP program. I suppose this indicates I misspell like most misspellers. The words FORMATTING, FLOPPIES, and MISSPELLERS were not in the dictionary. Of course, I *knew* how to correctly spell all the words. There were four easy typo's and I purposely made mistakes for timing considerations only.

As a test, I subjected two text files that were written over a year ago to the SPELL program. The first was an article I wrote reviewing a popular word processor. I had reviewed the manuscript several times and was pretty sure of its accuracy. The second file was a university term paper written by my wife. The paper received a grade of "A". It was written for a Technical Writing course and spelling was considered a prime area of evaluation. It was returned with no errors detected. The SPELL program found three misspelled words in the term paper and seven misspelled words in my article!

I enthusiastically recommend The WORD to all serious writers. Manuscripts, term papers, and official documents will always look just that much better with no spelling errors. The versatility of the programs make The WORD simply a set of writing tools and not a cure-all, do-all-for-everybody, turn-key, single program. Like any useful tool, the user must understand the limits and the designed purpose of the tool.

The WORD is not tutorial. While it is quite friendly and easily learned, it requires time and effort to become familiar with the many features found in the programs. For me, The WORD has become a necessary extension of my word processor. ▶▶J

Universal Data Systems 103LP Modem
Universal Data Systems
5000 Bradford Drive
Huntsville, AL 35805
Reviewed by Dave Caulkins

The UDS 103 LP is a 300 baud RS232 interface, answer/originate line-powered FCC Part 68 certified modem. The most important of these are the FCC certification and line-powered features. FCC certification means the 103LP can be directly connected to the phone line via a standard modular jack; direct connection improves the performance of the modem considerably over that possible with acoustic coupling. "Line-powered" means that the modem draws the power it needs from the phone line; there is no need for a 110V AC connection. This is a neat feature and as far as I know unique to UDS. It means you don't have to go hunting around for an AC outlet that's near the telephone jack.

The modem is quite small and light; it's designed to fit under a standard telephone. Installation is very easy; connect your terminal or computer to the RS232 female connector on the back of the modem, plug a standard telephone instrument into the modular jack labelled "TELSET" on the back of the modem, plug the telephone line into the modular jack labelled "TELCO" and you're ready to go. Set the switch on the back of the modem to "TERMINAL". If you're originating the call, set the switch on the back of the modem to "ORIGINATE". Make sure the switch on the front of the modem is set to "TALK", then dial the number of the answer modem with which you wish to communicate. When you get the modem answer tone, change the DATA/TALK switch to DATA; the LED should come on. At this point you can hang up the phone and start data transfer. When you've finished transferring set the DATA/TALK switch back to TALK to hang up the phone.

We've used the UDS 103LP to talk to a wide variety of computer systems and never had a lost or garbled character. The one problem we've had has been with the line-powered feature – there are some places in Menlo Park where the phone company evidently does not meet its own spec for central office battery, the power source needed by the UDS 103LP. It's hard to argue with the phone company about matters that don't relate to plain old voice service; if you get a UDS 103LP, make sure the company that supplies it to you will either allow you to return it or be prepared to deal with the phone company in case you are in one of those rare locations where the 103LP won't work. ▶▶J

SPECIFICATIONS

The UDS Model 103LP is a fully Bell-compatible, originate only modem with the following operating characteristics:

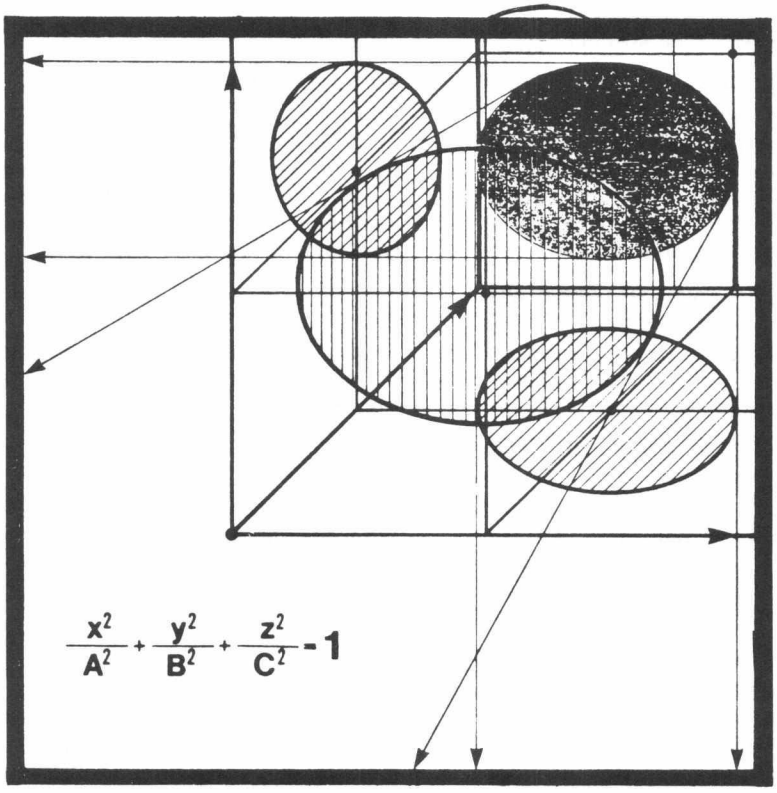
Operation – full duplex on two-wire circuit
Data Rate – 0 to 300 bps asynchronous
Modulation – phase coherent, frequency shift keyed (FSK)
Operating Mode – Switch-selectable, originate or answer
Line Type – DDD network
Carrier Frequencies – Compatible with Western Electric 300 bps modems
Line Impedance – 600 ohms
Transmitter Level – -9 db
Receiver Sensitivity – -8 to -46 dbm; meets FCC requirements, Part 68
Digital Interface – RS-232C, CCITT V.24, TTY
Controls and Indicators – Data/Talk switch; data ON light emitting diode
Originate/Answer Switch
FCC Registration – Model 1031LP; Registration Number AK396F-67636-DM-E; Ringer Equivalence 0.0B; Data Jack RJ-11C
Package – stand-alone

DR. DOBB'S JOURNAL

For Users of Small Computer Systems

$2.50
($3.00 in Canada)

Number 66 April 1982



8080/8086 Cross Assembler Part II
Jonathan Mills

Son of Runic
Marty Franz

Small-C DISKDOC
Egil Kvaleberg

and much more!

A Poor Person's Spelling Checker . . .

. . . Check Digits Maximized . . .

. . . Elliptical Algebra

DR. DOBB'S JOURNAL
For Users of Small Computer Systems
April 1982 Volume 7, Issue 4

CONTENTS

16 8080/8086 Cross-Assembler, Part 2
by Jonathan Mills

Last year's most widely acclaimed *DDJ* article was part one of this series, which covers the activities of the Z-88 Project. The author's latest contribution covers some features of the XASM-86 development language and of Intel's 8087 Numeric Data Processor chip.

26 Small-C DISKDOC: A Repair and Maintenance Utility
by Egil Kvaleberg

A down-to-earth addition to the toolkit of anyone who uses and prepares programs for diskettes. This program is written to be compiled by Ron Cain's Small-C compiler (*see our back issues, esp. #45 and #48*) and is meant to be used in a CP/M environment. Its appeal is even broader in scope, as it is written in a high-level language and has a general, easily adaptable structure.

36 Making the Most of Check Digits
by Peter Zilahy Ingerman

Check digits – once you know how to use them – can be much more than just simple error indicators. They can tell if there was one error or several, what the errors most likely were, and the probable correction.

42 A Poor Person's Spelling Checker
by Alan Bomberger

New proofreading and spelling correction programs seem to appear every month. Dictionaries contain more words, and editing functions become (somewhat) less cumbersome – yet most still leave much to be desired. This version is published to foster wider interest in the features and foibles of such software. Can anyone find a better way?

54 Son of Runic: Writing the Compiler
by Marty Franz

When this author opted for CP/M over his older Pet, he found the environment he needed for language development. This article describes some of the implementation details of his Forth-like Runic, an interactive, extensible compiler described in last month's issue.

4 Editorial

4 Letters to the Editor

Wherein readers correspond, despond, describe and diatribe over their upsets, subsets, sobriquets and sob stories.

9 Dr. Dobb's Clinic

Among the many cases and cures at the Clinic is a special patient; help our Intern ease the hyperbole!

12 Of Interest

As usual, the tides of industry brought bushels of press releases to our editorial shores. The foremost amid the flotsam can be found in this column . . .

58 Software Reviews

59 Hardware Review

60 Book Reviews

Editor – Marlin Ouverson
Contributing Editors –
Dave Caulkins, Dave Cortesi
Marketing Director – Craig Harper
Marketing Assistant – Ari Taylor
Advertising Sales – Laird Foshay
Circulation – Gloria Romanoff,
Theresa O'Rourke, Terri Marti
Art Director – Clifford West
Production Manager – Barbara Ruzgerian
Typesetter – Renny Wiggins
Proofreader – Nancy Heubach
Cover Illustration – Al McCahon

© 1982 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $40 for first class to Canada and Mexico; $58 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

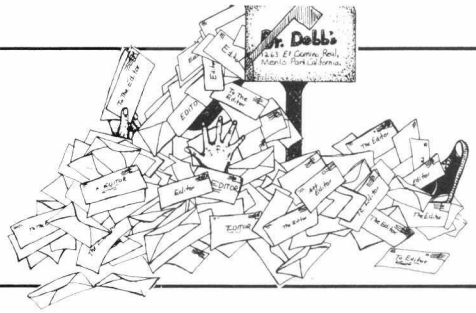
Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Eric H. Johnson, Jerry Horn, Edward W. Carr, Steven Fisher, Thomas H. Grohne, G. V. Elkins, Robert M. Connors, William J. McEown, Robert C. Luckey, Kim Leary, Robert N. Stabler, DeWitt S. Brown, John B. Palmer, E. Glion Curtis, Jack Miller, Burks A. Smith, Transdata Corproation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Howard E. Decker, Jim Wilson, Frank Lawyer, Chris Pettus, Paul Lutus, Ben Goldfarb, Rodney Black, Steven Weiss, Pat Phelan, John Brodie, James Fleming, Unison World. **Sustaining Subscribers:** Gordon Eubanks, Jr., and Pete Roberts. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** LP Enterprises, 8/11 Cambridge House, Cambridge Road, Barking, Essex, 1G11 8NT, England (Tel. 01-591-6511, TLX 892395). Hofacker-Verlag, Tegernseer Strass 18, D-8150 Holzkirchen, West Germany. **Asia & Australia:** Computer Store, POB 31-261, 22B Milford Road, Milford, Auckland 9, New Zealand. ASCII Publishing Co., Ltd., Aoyama Building 5F, 5-16-1 Minami Aoyama, Minato-ku, Tokyo 107, Japan.

Dr. Dobb's Journal (UPSP 307690) is published twelve times per year by People's Computer Company, Box E, 1263 El Camino Real, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025.

LETTERS



Back To BASIC Bugs In Santa Barbara

Dear Dobbs,

Thank you for your letter of the 22nd January concerning my query about SBTB in issue #55. From the other letter you enclosed it must seem that 6809 owners are a wild-eyed bunch given to such strange practices as hand entering object code (probably using toggle switches). Fortunately this is not the case, since this 6809 system supports a good (but not great) Macro Assembler, Son of EMACS full screen Text Editor, NROFF-style Text Processor, Sort/Merge package, Debugging facilities based on a full software emulation of the '09 processor, Extended BASIC interpreter, Pascal Compiler and more. Sort of what you find on S100 machines, but faster.

Now, on to SBTB, and its ills. I had noted the bugs referenced in the letter from Mr. Barton, plus others. Enumerated, they were:

1. Macro definition of HERE would not assemble.
2. Macro expansion of TSTC produces assembly errors.
3. No "not equals" expression in the expression analyzer.
4. The "less than" (<) comparison operator not a signed compare.
5. Unary Minus function not coded properly.
6. LIST command would only list the first line of the program.

To correct the HERE macro, make the definition of the first FCC pseudo op read "&1". The quotes are part of the definition.

To correct the TSTC problem, at every substitution instance of TSTC in the source code, make the compare quantity a quoted string, as TSTC "' ;",FI1. The TSC Macro-Assembler requires the quoted string, and I do not know how the assembly listing published was accomplished in a 0 ERROR(S) DETECTED condition.

The no "not equals" bug requires changing the definition of HERE '=' on page 26 of the original. Make it read HERE '#' and make the BEQ op code read BNE.

Similarly under HERE '<' make the BLS read BLE to create a signed compare.

The Unary minus problem may be corrected on page 27 by making the code read:

```
COMA
COMB
ADDD #1
```

Strangely, this 16-bit two's complement was coded properly later on in the code.

To correct the LIST problem, replace the TSTA instruction found six lines from the top of page 31 with a LDD ,Y which will then pass the next line number of the FNDLNP routine, allowing it to indeed find the next line pointer.

At this point, I have given at least first level testing to all the keywords and I am satisfied that they will perform nominal case actions. One final correction, from the text of the original PATB article, the abbreviation for "RETURN" is "RET.".

If anyone is interested in an all-FLEX version of SBTB, I will be happy to provide a listing in exchange for postage.

Very truly yours,
Douglas K. Beck
995 Lundy Lane
Los Altos, CA 94022

EDITORIAL

Wanted: Silicon Visions

Clandestine meetings to stifle foreign competition in the computer marketplace? Some industry leaders fear Japan is planning to repeat in Silicon Valley what it has done to Detroit. Unlike the cumbersome Motown giants, though, they aren't stubbornly content with roles as sitting ducks on a target range. They see the danger and aim to meet fire with fire.

The whole concept revolves around a fifth generation computer, destined to replace current technology in the 1990's. (Ha! Don't bet it will take that long . . .) Who will be the first to produce it, and just what will it be? Some computer and semiconductor firms are considering a joint venture to ensure that whatever it is, it won't be Made in Japan. I prefer to take the optimistic view, that this is an experiment in cooperation and won't end in monopoly. Although, there is that adage about power and corruption . . .

While the lawyers worry about anti-trust laws and Board members scrutinize profit margins, the international community of *DDJ* readers could set the pace by writing of their own ideas about the next generation of computers. Send in your silicon visions – regardless of country of origin – and we will get them in print as space permits. We will carry the spirit of cooperation to a global scale. There will be no editorial value judgments: be as ingenious, reasonable, outrageous or droll as you like.

Instead of independently re-inventing the wheel for clonish new products, the thought of foreign companies making serious inroads into U.S. profits may motivate the industry leaders to lend an ear to the innovators. It's time the mainframe makers began listening to the micro community without smirking! Ours is a younger, fresher viewpoint, a trait that has built more than one empire, in more than one nation.

Think about the concept of cooperation and about the attitude that anything is possible and should be tried at least once. They begat personal computing, and are the parents of *DDJ*. If we carry on the tradition a while longer, who knows? As long as we are thinking big, let's think BIG . . .

Martin Ouverson

Editor

The Eve of ADAM

Dear *Dr. Dobb's*:

You can pick up any issue of the SIGPLAN notices nowadays and see statements like

> Subsetting [of Ada] has become so necessary that it is administratively forbidden.

What is the penalty for writing a compiler for an Ada subset? Will the Pentagon dispatch an Ada Compliance Death Squad (ACDS) to your house? Do you want to take that chance?

Just to play it safe, I recommend that all Ada-like languages be given slightly different names. I am placing the following words into the Public Domain to be used to identify languages derived from Ada and for languages derived from other sources but dressed up to look like Ada in order to be fashionable.

A Partial Solution to the Ada Problem

ADACTYLOUS	GRADATION
ADAGE	GRANADA
ADAGIO	HAGGADA
ADAM	HEADACHE
ADAMANT	LACKADAISICAL
ADAPT	MACADAM
ARMADA	MADAM
CADAVER	NEVADA
CICADA	NOWADAYS
DEGRADATION	PLEADABLE
GADABOUT	RADAR
GONADAL	READABLE

Other words can be derived from these, and I think it is especially instructive that ADAMANTINE is almost a synonym of STEELMAN. Notice that some of the Ada-like languages might be well suited for military applications, particularly ARMADA and RADAR. My personal interest is in GADABOUT, which will be an Ada-like language that can compile and run on a 6502.

Douglas Crockford
14541 Countrywood Lane
Irvine, CA 92417

H8 Feature Not Quirky

Dr. Dobb's,

I noticed an item of interest in the November, 1981, issue of *DDJ*. It may already have resulted in a reply from another reader, since I admit that I got this issue from the "newsstand" rather late. But if it is still of interest, here goes.

Terrence Hills of Ontario wrote in about a "dangerous quirk in the Heath H8" front panel software monitor. The "quirk" results from a "feature" of the monitor, which is also shared by the monitor in the H88/H89 computer, namely HALT detection. The monitor attempts to detect the CPU's execution of a HLT instruction and exit to the monitor if found. As you described, the monitor does this by checking the opcode at the return address -1 for equality with 166 Octal (76 Hex), the value of the HLT instruction. As you pointed out, there is the possibility of other opcodes and data bytes trailing with a 166Q value that the monitor might be fooled into thinking is such an instruction. In theory, there is a risk of re-entry of the front-panel monitor every 2 milliseconds, when the clock interrupt occurs!

In practice, this risk is quite small. I have used the H88/H89 computers for several years and never had an unexpected re-entry to the monitor because of HLT detection. The reason would seem to be easily explained, it turns out. Consider the worst-case chance of a random 166Q byte being inadvertently decoded as a HLT. In the 2ms interval from one clock interrupt to the next, with a 2MHz clock, there will be 2,000,000/500 or 4000 clock cycles (assuming that the clock interrupt routine takes zero cycles – remember this is a *worst case* analysis). Of the opcodes available that could possibly contain a 166Q byte, the slowest 8080 code would be CALL xxxx where xxxx was an address having 166Q as the high byte; or the Z80 codes, the slowest ones ending in 166Q would be the INC (IX+166Q) , INC (IY+166Q) , DEC (IX+166Q) , and DEC (IY+166Q). The 8080 CALL requires 20 clock cycles, and the Z80 INC/DEC instructions on the index registers requires 23 clock cycles. This gives a worst-case chance of 20/4000 (0.5%) or 23/4000 (0.575%) that one of these opcodes would randomly fall just at the time of a clock interrupt.

Naturally, the chance of erroneous HLT detection is markedly increased if the CPU spends a lot of time executing opcodes with 166Q's in them, such as in a tight loop. Consider the potentially meaningful program in listing one. In that extreme example, the CPU spends only14 clock cycles of each loop iteration totaling 68 clock cycles, executing an instruction that does *not* end in 166Q! This loop has a 54/68 chance (79%) of landing in the monitor at any 2ms clock tick! Obviously, one would not want to use such code in an H8 or H89. It would have a 99% chance of bombing out within 6 milliseconds!

But that is not strictly true! Heath software engineers weren't way out in left field when they wrote the ROMs for the H8 and H88/H89. They recognized that software HLT detection would be a handy way for user programs to return to the monitor, but also a potentially great liability with certain program sequences. So they built in the ability to disable HLT detection, and made it an automatic feature invoked every time that their own disk operating system, HDOS, is booted up! This means that no H8 or H89 program running under HDOS is at risk of accidental monitor re-entry from "pseudo-HLT" unless the code specifically re-enables HLT detection!

As I have said, I've never had an accidental "HLT" with the cassette tape software provided by Heath (which uses HLT detection) nor, naturally, with HDOS disk software (since HLT detection is suspended automatically). But for the user who wishes to "guarantee" freedom from accidental monitor re-entry, it is quite simple. Listing two shows a simple piece of code that disables HLT detection in assembly language (and also shows subsequent re-enabling of HLT detection and re-entry of the monitor as Heath intended). Listing three shows the same functions in Benton Harbor Basic, and listing four shows Microsoft Basic (but MBASIC is only available on a disk system, and as I said, HLT detection is already disabled for you).

Hopefully, I have put reader Hills (and other H8 and H88/H89 owners) at ease about this potential problem. An experienced reader of 8080 assembly code can actually see how HLT detection words (and how it can be disabled) by examination of the H8's PAM-8 monitor code listing (or the H88/H89's MTR-88 ROM), which Heath provides free of charge with any of their cassette or disk software operating systems. Proof of HLT detection disabling under HDOS comes from inspection of the source code for HDOS, which can be purchased from Heath for $195.00. I have inspected both quite closely, and can state without reservation that the Heath software is solidly dependable and well-written, and in fact

```
ADDRESS OPCODE BYTES LBL    OPCODE  OPERAND  COMMENTS                      CLOCK CYCLES
166.166 333 166     LOOP    IN      166Q     ;get a byte from port 166Q          10
166.170 346 166             ANI     166Q     ;mask out unwanted bits              7
166.172 300                 RNZ              ;got a value, so return to caller    5
166.173 315 201 166         CALL    NOTYET   ;deal with "no input" situation     20
166.176 303 166 166         JMP     LOOP     ;and go back to try again           10
166.201 076 166     NOTYET  MVI     A,166Q   ;set limit in A                     7
166.203 270                 CMP     B        ;test against B                     4
166.204 300                 RNZ              ;return if limit not reached        5
                                                                              --------
                                                                 Total =        68
```

LISTING ONE: Section of code highly susceptible to PAM-8/MTR-88 HLT detection.

```
.MFLAG  EQU     040010A         ;monitor mode flag byte address
DISAB   LDA     .MFLAG          ;get flag byte in A
        ORI     200Q            ;set the  "U0.HLT bit"(200Q or 10000000B)
        STA     .MFLAG          ;put flag byte back, and HLT will now be ignored!

.
.
.
EANB    LDA     .MFLAG          ;Get flag byte
        ANI     177Q            ;Strip off "U0.HLT bit"
        STA     .MFLAG          ;put back flag byte, adjusted (enable HLT detection)
        EI                      ;make sure the 2ms clock can tick! (enable CPU interrupts)
        HLT                     ;Halt, let PAM-8 or MTR-88 take over.
```

LISTING TWO: Disabling and re-enabling HLT detection by Heath Monitor, in assembly language.

```
65000 FØ = PEEK(8200)   : REM Get monitor flag byte
65010 FØ = FØ OR 128    : REM Make sure high bit is set
65020 POKE 8200,FØ      : REM Store newly-adjusted flag.  HLT detection is now gone
65030 RETURN            : REM All done

----

65000 POKE 8200,(PEEK(8200) OR 128) : RETURN : REM Same as above, compacted.

----

65000 FØ = PEEK(8200)   : REM Get flag byte
65010 FØ = FØ AND 127   : REM Remove the high bit
65020 POKE 8200,FØ      : REM Store adjusted flag byte.  HLT detection now active
65030 RETURN            : REM All done

----

65000 POKE 8200,(PEEK(8200) AND 127) : RETURN : REM Same as above, compacted

----
```

LISTING THREE: Disabling and enabling HLT detection in Benton Harbor BASIC.

```
65000 'Disable HLT detection by PAM-8 or MTR-88 ROMs
65010 MF = &O040 * 256 + &O010      'Split-Octal address of .MFLAG converted to decimal
65020 POKE MF,(PEEK(MF) OR 128)     'Re-Store with high bit set
65030 RETURN                        'HLT detection is now gone

----

65000 'Enable HLT detection by PAM-8 or MTR-88 ROMs
65010 MF = &O040 * 256 + &O010      'Split-Octal address of .MFLAG changed to decimal
65020 POKE MF,(PEEK(MF) AND 127)    'Re-Store with high bit cleared
65030 RETURN                        'HLT detection is now re-enabled

----
```

LISTING FOUR: Disabling and enabling HLT detection in Microsoft BASIC

contains even more "hidden" features that many users are unaware of!

John P. Toscano
614 Portland Mews
Burnsville, MN 55337

(More on this aspect of the Heath computers may be found in this month's Clinic. – Ed.)

Bracketeer?

Dear Sir:

My letter of complaint concerning the Heath Company, which you saw fit to print in issue 56, drew two responses. My first response came from a member of the Heath User's Group. Unfortunately, it caught us as we were relocating from Massachusetts to Colorado and was lost before I had a chance to answer it. The writer informed me that Heath, out of the goodness of their collective heart, refused to sell me the brackets because mounting a drive in the terminal enclosure is not such a good idea anyway and is doomed to cause problems. The writer chose to ignore Heath's eagerness to sell the brackets – complete with a $230-Siemens drive – to anyone willing to part with $450.

The second response appeared in your issue 60. Mr. Shoemaker, who seems to be a real Heath fan, states that the required parts are stocked in "large numbers" by the Alexandria Heath Centre. A call to the folks in Alexandria revealed that these parts are not in stock. However, Mr. Shoemaker included in his correspondence a list of the part numbers (Part 206-1331, a Mu-metal shield, is now 206-1346), and for this I am very grateful. A call to Heath produced an actual order, and for $31.80 I now have a drive mounted in my H-19 terminal. One last note: Heath will not accept a charge order for parts, although they just LOVE to charge for instruments. To summarize: my feeling about Heath has not changed. Anyone who plans to buy anything from that company, should know what after sale service to expect.

Sincerely yours,
R. A. Hoffman
1341 Ginny Lane
Colorado Springs, CO 90818

DDJ

DR. DOBB'S CLINIC

by D. E. Cortesi, Resident Intern

The Continuous H8

In the November Clinic (*DDJ* #61) we noted a complaint from Terence Mills. He'd found that PAM, the panel monitor of his H8 took a timer interrupt every 2ms. While processing the interrupt, PAM checked the interrupted instruction; if it appeared to be a HLT, PAM would return to console mode under the assumption that the program had quit. Trouble was, Terrence said, there are instruction sequences that look like a HLT to PAM but aren't.

Two readers, Edgar Howell of Simi, CA, and David Shaw of Elmhurst, IL, sent clarifications. Shaw's gives the most detail:

"Actually, this is a feature that can be quite useful when entering and debugging programs from the front panel. As described on page 16 of the PAM-8 reference guide, a halt instruction may be inserted where a breakpoint is desired. When the program gets to the breakpoint, control is returned to the PAM-8 monitor. Pressing the GO key will continue program execution.

"This feature and several others are under programmer control by setting and clearing bits in the .MFLAG byte. You can, for example, stop PAM-8 from updating the front panel display, turn the display off altogether, or request clock interrupt service. Other control flags allow you to turn off the 2ms clock, turn on the built-in speaker, and enable single-step hardware. All of these features are documented in the PAM-8 source listing, Appendix A of the reference manual.

"Setting the UO.HLT bit will inform PAM-8 that breakpoints are not wanted:

```
.MFLAG  EQU  40010A
UO.HLT  EQU  10000000B
        LDA  .MFLAG
        ORI  UO.HLT
        STA  .MFLAG
```

HDOS turns on UO.HLT when you boot up, so you don't have to do it yourself unless you are programming through the front panel; I assume that Heath CP/M does the same thing."

ED Finds Them All

Many people buy a full-screen editor with their CP/M system and so never really learn the ED editor that comes with it. ED isn't a very elaborate editor – we wouldn't touch it for writing purposes – but it has its uses as a utility. One of its charms is its Macro command, which allows you to execute any sequence of commands repeatedly.

W. G. Hutchison, Jr., of Exton, PA, sent us a "sponse" in this regard. "Here is a way to generate special purpose indexes of programs using ED. Use the following command string:

```
B#MFstring^Z0L1T1L
```

This can be used to list all function definitions in a CBASIC program (*string* is 'DEF'), to list all function calls (*string* is 'FN'), to list all uses of integer variables (*string* is '%'), etc."

If you haven't explored the use of Macro, you'll enjoy working out what Hutchison's command string does. There ought to be a lot of clever things that can be done with Macro; what others can you find? When driven by a script of commands via SUBMIT and XSUB, ED has considerable power as a file-manipulation utility. What uses can you find for that combination?

Robotics Help Needed

Ray Poli of Flint, MI, needs a line on information sources. He writes, "Could you please help me get started writing programs to control a robot I've built? I can't seem to find any literature on the subject. I have a 64K H8 with CP/M, and can use ideas, suggestions, etc."

It seems clear that Ray needs general background information, and also needs contact with other experimenters in robotics. We've heard of one source. *Robotics Age* magazine, a recent press release tells us, has been bought by a New Hampshire firm and will be edited by Carl Helmers, once founding editor of *BYTE*. Under the new management, "Heavy emphasis is placed on practical tutorial information." The magazine costs $15 a year; its address is 174 Concord Street, Peterborough, NH 03458.

That's the only publication we know of, but surely the readers know of more. Tell us the books and magazines that have good info on robotics, and let us know the addresses of the experimenter's organizations. We'll print a list here.

An Elliptical Query

Don Taylor, of Corvallis, OR, has a simple question. You'd better hear it in his own words, because we haven't a clue:

"I am looking for the algebraic formula for an arbitrary ellipsoid, rotated in all three axes, projected on the X-Y plane. In other words, given

$$\frac{X^2}{A^2}+\frac{Y^2}{B^2}+\frac{Z^2}{C^2}=1,$$

rotate it around the X axis D degrees, then the Y axis E degrees, and then the Z axis F degrees. Take the resulting ellipsoid and find the projection on the X-Y plane.

"I believe the resulting curve is an ellipse, but am unable to prove that. I have attacked the problem in several ways. One method was to use rotation of axis substitutions to obtain the general formula for the oriented ellipsoid, and then consider the formula as a quadratic in Z. The projection consists of all points that have a single value for Z, and this implies that the discriminant must equal zero.

"The number of terms involved precludes hand computation, so I wrote a small algebra handling program in ALISP and ran it on the local Cyber. I got as far as finding the a, b, and c for the discriminant's $b^2 - 4ac$. The supervisor in ALISP informed me that I had run 14 garbage collections, was in control of 80% of core, and did not have enough room to complete the calculation. The program was trying to multiply 25- to 40-term polynomials, each with three variables, and sines and cosines of three angles.

"There should be some elegant method of finding the formula of the solution, but I just don't see it. It's been suggested that I solve the problem in spherical coordinates and translate the solution back to cartesian, but the initial work didn't seem to be much better. Use of matrices and performing a change of basis was suggested, but the equations are not linear. Locating four extreme points and solving for the elipse that goes through all the points might be a possibility. I had some luck with this method until I found I was not solving what I thought I was.

"Now that I have blown you away with what I have tried to do, HELP!"

Touch-Tone ASCII

Several readers responded to our query for a simple way to encode alphabetic input through a Touch-Tone keypad ("Touch-Tone," by the way, is a Bell trademark). Everyone observed that most

of the alphabet is already on the keypad, three letters per key. There are three problems; how to shift from numbers to letters, how to signal which of the three possible letters is meant by a particular key-press, and what to do about Q, Z, and the blank.

The usual solution to the third problem is to pretend that Q, Z and blank are printed on either the zero- or the one-key, which don't have any letters. Then those letters can be handled in the same way as the others.

Andrew Sackheim proposes that "to enter letters, you first enter '*', then the number the letter appears on, then the digit corresponding to the relative position of the letter (1, 2, or 3), and finally an optional '#'. The word 'help' would be entered as '*42*32*53*71#.'"

Bruce Komusin of Huntington, NY, proposes a variation on that theme. He notes that the keypad has a left special key, "*", and a right special key, "#". He'd use these to mean "left letter" and "right letter" respectively. He also suggested that pressing "*#" in succession should mean "middle letter," but that is redundant. If a letter is neither left nor right, it has to be middle. So, presuming the system knows you intend alphabetic input, the modified Komusin code would give the world "help" as "435#7*." If you have to tell the system which kind of input is coming, the sequence "##" could be used for "numeric shift," and "**" could be used for "alpha shift." This code yields the shortest key sequences of any general scheme proposed.

W. Hutchison notes that in certain cases, no special coding is needed for the letters. When the system expects a response chosen from a limited vocabulary of keywords, the user can press the appropriate letters and the system can figure out which keyword is meant. All that is needed is to choose the vocabulary of keywords so that there are no abiguities. For instance, you couldn't have both "cow" and "boy" in the list because they would both be spelled "269."

Dan Dugan of San Francisco, CA, brought up an important point. "The Bell System special features offered by newer exchanges (call-waiting, call-forwarding, etc.) have established the "#" key as having an ENTER or RETURN function." If you want to be compatible with the Bell conventions, you should reserve the "#" to be a delimiter or terminator. How important is that?

Dan Samuel of Palo Alto, CA, suggested a variation on the same code proposed by Sackheim. But he had other news, too. We'd asked for a design for a complete Touch-Tone interface for a small computer. Steve Ciarca addressed the design basics in an excellent article in the December, 1981 issue of *BYTE*, but Samuel's company, Digital Pathways (1060 E. Meadow Circle, Palo Alto, CA 94303), have gone further. They make the SLC II, a unit that "attaches to the phone line and provides full 16-tone encoding and decoding as well as synthesized speech output over the phone." It can be driven as a peripheral via an RS-232 interface, but it is a computer itself, having a 6502 CPU, 80KB of banked storage, BASIC, a clock and calendar, and four serial ports (but no disk interface). So the "world's smartest phone-answering machine" that we asked for already exists.

Pass The Hat

We have our reply mechansim in place now; we'll start an acknowledgement your way within two weeks of receiving your letter. You know, the letter you were just about to write, with a question, a discovery, an algorithm, or a problem. ▶▶J

OF INTEREST

by D. E. Cortesi

Tandy's New Machines

In January, Radio Shack announced three new products, each a highly significant machine in its field. Their 16-bit desktop machine, video terminal, and enhanced pocket computer will each make waves in the industry. Taken together, they show that Tandy Corp. is a computer company that doesn't have to apologize to anyone.

The new Model 16 is most exciting. It is a desktop machine similar in appearance to the TRS-80 Model II. Where the Model II had a single 8-inch diskette drive, the Model 16 fits a pair of the new, slim, double-sided drives, giving it 2.5MB of disk capacity within its case.

Inside, the Model 16 is to have 128KB of RAM (expandable to 512KB) and a pair of CPU's: a Z80 and an MC68000. The early press releases are not at all clear on the relation between these two processors. The Z80, it is said, "relieves the MC68000 of time-consuming housekeeping chores," but it can't be limited to that because the presence of the Z80 allows "software compatibility with existing Model II software." We wonder how they accomplish the switch-over from running Model II software on the Z80 to running new software on the MC68000. Regardless, the Model 16 is said to be powerful enough that it can support two additional terminals. "This lets three users access programs and information at the same time, with no perceptible loss of performance" – an interesting claim.

The Model 16 is to sell for $5800 with two disk drives and 128KB. This puts it one up on the much-touted Fortune 32:16, which has only a single 5-inch drive at that price. The MC68000 CPU, big drives, multi-user ability, and compatibility with existing programs all work to put the machine ahead of the IBM Personal Computer, and at a price that isn't much greater. A $1500 upgrade kit will be available to add the 16-bit CPU, 128KB of storage, and multi-user TRSDOS to existing Model II machines.

Radio Shack also announced the DT-1, a 24-line by 80-column video terminal, with a price tag of $700. It will emulate any of four popular terminals and supports reverse video, blinking, underline, and low-intensity display fields – but not graphics. It has a standard typewriter keyboard and both serial and parallel interfaces. Its configuration parameters are set from the keyboard and are remembered over power-down.

As if those weren't enough, an enhanced version of the Pocket Computer was announced as well. The PC-2 has an extended version of BASIC in a 16K ROM, its keyboard has been rearranged into typewriter layout, and its display now supports lowercase letters and graphics. Its initial 2600 bytes of programmable storage can be upgraded to 16KB in 4K increments. A four-color printer (!), two-drive cassette interface, and RS-232 port are promised. The PC-2 brings us spookily close to the kind of pocket secretary that is a standard prop in contemporary science fiction. All that's lacking is a multi-line display and pocketable mass storage. Can they be far off? **(Reader Service No. 301)**

A Significant Decision

Much has been written about the attempts by Atari and others to lock up the rights to computer games. These disputes are not over the implementation of a game – but over the rights to the game's situation and its appearance on the screen. Atari, for instance, has moved against some of the entrepreneurs selling clones of Pac Man for different personal computers, despite the fact that the programs involved are the work of the sellers.

Now the U.S. Court of Appeals for the Second Circuit has made a significant ruling in this area. The case involved the popular "Scramble" arcade game, which was licensed to Stern Electronics of Chicago. A company called Omni Video Games brought out a very similar game, using their own software. Stern asked for an injunction, which was granted and which has now been upheld by the Appeals court. The way "Scramble" was registered for copyright is significant. Rather than registering the software that produced the game, a video tape of the game was registered as an audiovisual work. Omni Video Games protested that, since no two plays of the game will produce the same sequence of images, there was no single "work" to be copyrighted. The court disagreed, pointing out that the game's elements – the various spacecraft, the missiles, the terrain, the explosions, the sound – are constant, and therefore the whole comprises a "work" that can be copyrighted, even though the sequence of its presentation varies with the player's skill and luck.

This decision will add strength to Atari's attempts to control the marketing of Pac Man and other games. If you've had the notion to program a version of some arcade game for your Apple or Atari, perhaps you should think again. And if you come up with an original game idea, don't just register a program listing for copyright. Get a video tape recorder and register a game session complete as an audiovisual work. If this decision holds up, you might then own a game for *any* computer, not just your specific implementation of it.

Information on Tap

In our opinion, the greatest social impact of personal computers will come from the networking they allow – the connection of poeple to people and of people to information sources. There are already a lot of public databases to be tapped, enough of them that the tireless counters-and-sorters at **Datapro** have found it worthwhile to publish a new *Directory of On-Line Services*. They claim it's "the most comprehensive reference of its kind, with over 100 on-line services and 500 databases." As is usual with Datapro directories the information is categorized and cross-indexed to a fare-thee-well, and it carries their customary, stiff price tag: $390 for two fat binders and a year's monthly supplements. There's a publishing opportunity there for a competitor. **(Reader Service No. 311)**

One of the largest on-line systems is **Dialog**. They are about to add two databases of interest to users of personal computers. File 233 is to contain a microcomputer magazine index. It will have 6,000 entries initially; 1,200 more will be added each quarter. File 232 will contain a microcomputer software directory; its initial 5,000 entries are to be updated monthly. **(Reader Service No. 321)**

One of the oldest on-line systems is the tickertape service provided by the New York Stock Exchange. Its data are available through other systems, but with a 15-minute delay. Now, **Wall Street Software** has prepared a $500 hardware and software package with which an Apple II can connect directly to the NYSE's "A" tape. The hardware is a communications card. The software scrolls the ticker across the top of the screen, and it extracts quotes for the stocks you are interested in for display in a fixed area below. **(Reader Service No. 331)**

More Info, Asked and Given

Thinking of starting your own business? Of course you are; we all plan to, someday. **Wildfire Publishing** have prepared a two-volume work under the title *Your Fortune in the Microcomputer Business* which, they claim, is a complete manual for starting and managing your own business. Each volume costs $20. Volume I tells how to survey the market, select a product, get capital, and start the business. Volume II "deals with the problems and decisions a person faces when their business really takes off." A canny person might postpone buying that one until they're sure they'll need it. **(Reader Service No. 341)**

Sapana Micro Software is poised to leap upon the IBM bandwagon; they are collecting vendor information with a view to publication of a catalog of hardware and software for the IBM PC. Vendor entries are solicited. **(Reader Service No. 351)**

Users of the Nevada COBOL compiler may be interested in knowing that a **Nevada COBOL User Group** is now being formed. No dues are mentioned. **(Reader Service No. 361)**

Software, Sophisticated . . .

Here are three language items that caught our notice this month. **Curry Associates** have prepared a set of FORTH programming aids. One is a decompiler (dis-forth-er?) that will convert compiled FORTH words into structured source code, for documentation or revision. Another is a command to find the words called by a specific word to any level. It can be used, they say, to extract the set of words needed to compile a program of minimal size, or to examine the words called within a loop, the better to optimize the code. **(Reader Service No. 371)**

Last month we mentioned a structured pre-processor for BASIC. It's clearly an idea whose time has come, and now we've heard of another one. This one is BLX from **Quelo**. The programmer writes using a "rich assortment of structured programming statements with full nesting," and using names for subroutines. The pre-processor rewrites the program into a form suitable for either CBASIC or MBASIC. The pre-processor costs $65; a utility to convert existing programs into BLX-BASIC is available for another $30. **(Reader Service No. 381)**

The ADA language is big in certain circles, and **RR Software** moves in them. They publish a compiler called JANUS, which processes "an ADA subset." JANUS is available in 8080 form for CP/M, and in 8086 form for CP/M-86 and for IBM DOS. It costs $480. We think this product is "of interest," but we'd like to remind you that in one sense there is no such thing as a subset of ADA. The language spec includes no provision for subset implementations. Unofficial subsets are inevitable given the complexity of the language, but strictly speaking, no subset is really ADA. A subset can be at best "ADA-like," and couldn't be acceptable to the Department of Defense – if you care. **(Reader Service No. 391)**

. . . And Not.

Several nice little programs turned up this month. Among them was **Donald Fielding**'s WILDCAT. It's a $42 disk catalog system for the TRS-80 Model I, with versions for TRSDOS and three levels of NEWDOS. The cataloged information includes not only the file's name and vital statistics, but the first line of its contents as well, providing a sort of file-abstract function. **(Reader Service No. 401)**

Another is the 3D Drawing Board for the Color Computer, from **Computerware**. It aids the drawing, scaling, and rotating of 3-D objects on the screen, and allows the drawing to be saved on tape or disk; it costs $25 on cassette. **(Reader Service No. 411)**

A third is LINE#DATA, for the TRS-80 Model I with a disk, from **En-Joy**. This $10 program converts data into DATA statements. It will read the screen memory, or take strings from cassette or disk, convert what it finds into DATA statements, and save them on disk for later merging into a program. En-Joy also sell a clever TRS-80 screen template for $4. The clear plastic template is sized to fit exactly the layout of character positions on the screen. You build a picture on the screen, lay the template over it, and read the position or POKE indexes directly from it. **(Reader Service No. 421)**

Harder Ware

The "Grafyx Solution" from **Micro-Labs** is a $300 hardware mod for the TRS-80 Model III, with supporting software. The hardware gives the screen a resolution of 512 by 192 pixels (which dimension is up and which is sideways?). The software provides 80-column characters and the ability to save and load graphics screens or copy them to a printer. The hardware consists of an add-on circuit board with 12K bytes of graphics storage. Installation, in the words of the release, "is as easy as clipping on some micro-clips, cutting one trace, and removing two ICs from sockets." That easy? **(Reader Service No. 431)**

Users of the Osborne 1 are no longer constrained to buy Osborne's video monitor if they want to relax their squint. **Best Computer Stores** have prepared an adapter that will connect the machine to any standard composite video monitor. It's a $60 flat box that plugs into the front panel; they say you can close the machine's case while leaving the adapter plugged in. **(Reader Service No. 441)**

And we note that DEC have squeezed the "base level" instruction set of the PDP-11 into a single, 40-pin chip. The Micro/T-11, as they call it, should be available in sample quantities now. It will cost $75 per chip when ordered in quantity. Any bets on how soon we'll see one on an S-100 board? **DDJ**

Part II

8080/8086 Cross Assembler

Thank you, gentle readers, for your interest in the Z-88 Project, and specifically the XASM-86 Cross-Assembler (*DDJ* #56). Marlin Ouverson has relayed your enthusiasm, as have those of you who wrote me (but, alas, were not answered). I can only say that in most cases requests for source or object code on variously formatted disks will *not* be honored, because I have neither the time, the inclination, nor the appropriate disk drives to do so.

In this installment I will discuss some of the features of XASM-86, as well as share some exciting news about a superior arithmetic processing chip, the 8086 Numeric Data Processor by Intel.

XASM-86 is a development language tailored to my needs, and based on my past experience. This includes a range of computing equipment from Hewlett-Packard and Texas Instrument calculators to IBM, DEC and Burroughs minis and maxis. Languages include COBOL, FORTRAN, RPG II, BASIC, PASCAL, various assemblers and programmable calculator "lingo's" and, recently, database languages and specialty languages, such as NUFORM by the A.S. Thomas company.

All of this is mentioned to keep me from being branded as "crazy" by various persons in the computing community. I enjoy using these languages, and do not fanatically defend or condemn any of them. I do have some strong opinions on the *use* of programming languages, which have helped to guide the development of XASM-86. These opinions are:

Structured programs are the result of structured programmers. It is possible to write structured code in *any* language, as well as unstructured code. The difference lies in the training and motivation of the programmer. (If you don't believe me, and think that Pascal always produces structured code, examine the APL interpreter published in *The BYTE Book of Pascal.*)

Some languages have better tools to use to build structured code. While you can open a bottle of wine with an axe, it is much easier to use a corkscrew. In a similar fashion, those languages with clearly defined programming structures (sequence; if-then-else; do-while loop; and case-of-condition) allow the programmer to spend time concisely implementing his or her algorithm instead of wrestling with a language's deficiencies. Other tools that I feel are essential are:

- 15-character or greater variable names
- data declaration and typing
- easily applied "pretty-printing," such as blank lines between modules, and moderate indentation.

A language should help the user write, debug, maintain, and document the program. These goals tend to work against each other. COBOL can be made self-documenting, but it then becomes a horrendous task to write and debug the program. On the other hand, it is very easy to write and debug a BASIC program in the heat of creativity. Six months later, though, another programmer may think it extremely difficult to modify this same program, especially if the documentation is nonexistent! There is no happy medium here, unless it is the one practiced by assembly language programmers: write the code, and place the explanation next to it. Even this, by the way, can be carried to extremes. Examine a listing where every instruction is commented. It soon becomes bewildering.

XASM-86 is my attempt to create a development tool that includes the best of those languages in my background, avoids the worst, and is easy and pleasant to use. Since I am its prime user (along with others I have trained, but who were "naive" assembly language programmers) I felt no need to conform to any past practices that I detested, nor to cater to the inertia present in a large body of users who expected Intel standard mnemonics, or six-character labels, or other foolishness.

XASM-86 is new, and will set its own standards. Two of its features I fully expect will be found in other languages in the future. The first is the DITTO OPERATOR (") which was discussed in the last article. You will find it in the listings accompanying this article; those who have seen XASM-86 in operation frankly wonder why it hasn't been used before.

The second feature is less obvious (in some cases it is invisible!) but is just as useful. It is called SIGNIFICANT TEXT. Using this, it is possible to write assembly language programs that can be mistaken for Pascal programs, that are as self-documenting as COBOL programs, but are easy to key in and simple to debug. Some claims, right? Let's define significant text, and then look at a listing using it.

> SIGNIFICANT TEXT Upper-case letters and numbers used to define labels, mnemonics, and data; certain special characters used as operators and to control the assembly process. All other characters, such as lower-case letters, blanks, etc., and any significant text following the significance switch character – in XASM-86 this is a period (.) – will be ignored by the language processor, and dropped from the compiled or assembled output.

Now examine the source and object code shown in Figure 1. This short routine is purposely made into a very bad example of assembly language at its worst. If it were not for the header comments, it might have been a day or more before the purpose of this routine was discovered. However, there are no visually distracting parts of the code that would confuse a programmer trying to debug this routine. XASM-86 will accept this type of code, which is useful at times for quick test routines.

The listings in Figure 2 show the opposite end of the input spectrum handled by XASM-86. In Figure 2a we find significant text used (somewhat excessively) to illustrate its benefits. First, the source code is *very* well documented and commented. Note the use of operand equates that are combined with other, insignificant text to virtually duplicate the algorithm as it was written in English. The object code (Figure 2b) though, is quite different. It is sparse, having had all the insignificant text removed by XASM-86. What remains is useful when the routine is debugged, and has nothing in it to confuse the eye. Since I am using XASM-86 as the language for our commercial development project, some of you will decide that I must be a confirmed masochist. Nothing of the sort! XASM-86 is not demanding in its input expectations. Listings 3a and 3b show a typical program development to test the operation of our 8086 Numeric Data Processor exten-

by Jonathan Mills

Jonathan Mills, copyright © 1981 by Programmers Publishing Company, Box 2571 Kalamazoo, MI 49003

sion to the Z-88 CPU board. Notice that *both* types of text, and several commenting options have been mixed to create a self-documenting source. However, the object code is reduced to the bare-bone fundamentals! This one feature of XASM-86 allows the unprecedented production of reliable, maintainable software that is very large. Currently, our in-house project is scheduled for completion in late 1983, and will probably have over 90 Kbytes of object code . . . all written and maintained in XASM-86 assembly language.

This is as good a time as any to lead off with a discussion of the Intel 8087 Numeric Data Processor. Perhaps the best thing to say is that it is *incredible*! Imagine writing 11 assembly language instructions that will calculate the sine of a number to 18-digit precision, and convert the result to a binary-coded decimal number 10 bytes long in a millisecond (sorry to be so slow, but our sample only runs at 2 megahertz instead of 5 or 8). Actually, a library routine with data checking and scaling might take a little longer, but it is possible to have the calculation run concurrently with other portions of the routine, so I might very well be over-estimating the time.

This type of arithmetic is on a par with that found on an IBM 370 in its precision, and is superior to the instructions at the assembly language level available on *any* Digital Equipment computer I am familiar with, up to and including the VAX

Figure 1

```
;;
;
; 31 MAY 81
;
; THIS VERSION OF A SIMPLE SUBROUTINE TO CONVERT A BINARY
; NUMBER INTO AN ASCII DECIMAL DIGIT, WITH BINARY REMAINDER,
; USES THE LOWEST LEVEL OF SOURCE TEXT INPUT TO XASM-86.
;
; NOTICE THAT NO COMMENTS ARE PRESENT,
;             REGISTERS AND CONSTANTS ARE SPECIFIED BUT NOT NAMED,
;             ALL TEXT IS "SIGNIFICANT", OR MEANINGFUL TO THE ASSEMBLER.
;

DIGIT: XOR     DX,DX
       DIV     DI
       ADIB    30H,AL
       MOVB    AL,@BX
       MOV     DX,AX
       MVIB    30H,CH
       RET

;;
```

```
Z80-TO-8086 CROSS-ASSEMBLER   (C) 1981 BY  JONATHAN W. MILLS

VERSION      :  1.0
VERSION DATE:  08 MAR 81
EXCEPTIONS   :  MACROS UNDER DEVELOPMENT


ADDR CODE DISP DATA  LINE LABEL:.....OPCODE....PARAMETERS.....COMMENTS.....
                     0000 ;
                     0010 ; 31 MAY 81
                     0020 ;
                     0030 ; THIS VERSION OF A SIMPLE SUBROUTINE TO CONVERT A BINARY
                     0040 ; NUMBER INTO AN ASCII DECIMAL DIGIT, WITH BINARY REMAINDER,
                     0050 ; USES THE LOWEST LEVEL OF SOURCE TEXT INPUT TO XASM-86.
                     0060 ;
                     0070 ; NOTICE THAT NO COMMENTS ARE PRESENT,
                     0080 ;             REGISTERS AND CONSTANTS ARE SPECIFIED BUT NOT NAMED,
                     0090 ;             ALL TEXT IS "SIGNIFICANT", OR MEANINGFUL TO THE ASSEMBL
                     0100 ;
                     0110
0000 31D2            0120 DIGIT:          XOR       DX,DX
0002 F7F7            0130                 DIV       DI
0004 80C0      30    0140                 ADIB      30H,AL
0007 8807            0150                 MOVB      AL,@BX
0009 89D0            0160                 MOV       DX,AX
000B C6C5      30    0170                 MVIB      30H,CH
000E C3              0180                 RET
```

Figure 2a

```
;
;
; 31 MAY 81
;
; THIS VERSION OF A SIMPLE SUBROUTINE TO CONVERT A BINARY
; NUMBER INTO AN ASCII DECIMAL DIGIT, WITH BINARY REMAINDER,
; USES THE FOURTH, ARBITRARILY "HIGHEST" LEVEL OF SOURCE TEXT
; INPUT TO XASM-86.
;
; NOTICE THAT COMMENTS ARE INTEGRATED INTO THE SOURCE TEXT,
;             REGISTER & CONSTANT NAMES COMPLEMENT COMMENTS,
;             AND ONLY CERTAIN PORTIONS OF THE TEXT ARE "SIGNIFICANT".
;

REMAINDER:          EQUATE DX
DIVISOR:            "      DI
ASCII_CONSTANT:     "      30H
DECIMAL_DIGIT:      "      AL
@_RESULT_POINTER:   "      @BX
NUMBER:             "      AX
BLANK:              "      CH
ZERO:               "      30H


begin the DIGIT: module

this module will convert the highest .(leftmost) digit of any
decimal number from .0 to 65536, expressed as an unsigned binary
word.  the module places the decimal digit, adjusted to
.form an ascii character, into a user defined output buffer.  the
remaining digits will be expressed as the corresponding binary
number. they will be returned to the user for subsequent conversion
in .NUMBER.


     XOR the REMAINDER to the REMAINDER to make a .32-bit operand
     out of the .16-bit number to be converted

     DIVide the .NUMBER. by the DIVISOR, which contains a decimal
     scaling factor .(10,100,1000,or 10000)

     ADIB the ASCII_CONSTANT to the DECIMAL_DIGIT to make a
     printing character

     MOVB the DECIMAL_DIGIT to the byte in the output @_RESULT_POINTER

     make a new number to convert by MOVing the REMAINDER to NUMBER

     MVIB ZERO to BLANK to place zeros into the result
     instead of blanks after converting a non zero digit

     RETurn

.end the DIGIT module
```

Figure 2b

```
##

Z80-TO-8086 CROSS-ASSEMBLER   (C) 1981 BY  JONATHAN W. MILLS

VERSION      :  1.0
VERSION DATE:  08 MAR 81
EXCEPTIONS   :  MACROS UNDER DEVELOPMENT


ADDR CODE DISP DATA  LINE LABEL:...........OPCODE....PARAMETERS.......COMMENTS.....................
                     0000 *
                     0010 * 31 MAY 81
                     0020 *
                     0030 * THIS VERSION OF A SIMPLE SUBROUTINE TO CONVERT A BINARY
                     0040 * NUMBER INTO AN ASCII DECIMAL DIGIT, WITH BINARY REMAINDER,
                     0050 * USES THE FOURTH, ARBITRARILY "HIGHEST" LEVEL OF SOURCE TEXT
                     0060 * INPUT TO XASM-86.
                     0070 *
                     0080 * NOTICE THAT COMMENTS ARE INTEGRATED INTO THE SOURCE TEXT,
                     0090 *             REGISTER & CONSTANT NAMES COMPLEMENT COMMENTS,
                     0100 *             AND ONLY CERTAIN PORTIONS OF THE TEXT ARE "SIGNIFICANT"
                     0110 *
                     0120
0000          1200   0130 REMAINDER:       EQUATE    DX
0000          3F00   0140 DIVISOR:         "         DI
0000          3000   0150 ASCII_CONSTANT:  "         30H
0000          0000   0160 DECIMAL_DIGIT:   "         AL
0000          3F00   0170 @_RESULT_POINTE: "         @BX
0000          0000   0180 NUMBER:          "         AX
0000          2D00   0190 BLANK:           "         CH
0000          3000   0200 ZERO:            "         30H
                     0210
                     0220
                     0230
0000                 0240 DIGIT:
                     0250
                     0260
                     0270
                     0280
                     0290
                     0300
                     0310
                     0320
                     0330
                     0340
0000 31D2            0350                   XOR       REMAINDER,REMAINDER
                     0360
                     0370
0002 F7F7            0380                   DIV       DIVISOR
                     0390
                     0400
0004 80C0      30    0410                   ADIB      ASCII_CONSTANT,DECIMAL_DIGIT
                     0420
                     0430
0007 8807            0440                   MOVB      DECIMAL_DIGIT,@_RESULT_POINTER
                     0450
0009 89D0            0460                   MOV       REMAINDER,NUMBER
                     0470
000B C6C5      30    0480                   MVIB      ZERO,BLANK
                     0490
                     0500
000E C3              0510                   RET
                     0520
                     0530
```

series to date. The 8087 breaks the "number-crunching barrier" which formerly limited the microcomputer. What we will find on the other side will depend on the ingenuity of its users, but I expect a generation of superb low-cost graphics terminals for starters. The IBM personal computer should be able to host an 8087, at which time the 6502 machines will no longer be the fastest kids on the block. One enterprising individual I have come in contact with intends to mate the 8086+8087 duo to his 68000 add-on board for the Pet and Apple computers. Hmmmm.

Here at Programmers Publishing Company, we have successfully matched the 8087 to our Z-88. This little trick was somewhat touchy, since our 8088 was configured for the minimum (or very simple) mode, while the 8087 required that the 8088 run in the maximum (or multi-processing bus) mode. Although a possible solution was simply to rebuild the Z-88, it was not generally well-received. Instead, a circuit was designed that made the maximum mode 8088+8087 combo "look like" a single, minimum mode 8088. Voila! After much examination of bus loading, ringing, clock speeds, and data and control signal skew, we managed to build a daughter board for the Z-88. This board connects to the Z-88 via a 40-pin DIP jumper, which may not be elegant, but is at least efficient (ever tried to plug 40 wires together one by one?).

In any event, the 8087 sample has proved remarkably reliable. It has no opcodes that we have yet been able to observe that don't work; some of them work even *better* than Intel intended! For example, the partial tangent (FPTAN) instruction is valid on our sample from greater than zero radians to 1.35 radians, instead of only pi/4 radians as specified. Since we are not in the business of providing quality assurance for Intel, only those instructions we use are checked, but those we checked, worked. For a complete discussion of the 8087, the Intel Numerics Supplement to the 8086 Family User's Manual is essential. Application Note AP-113, "Getting Started with the Numeric Data Processor," is very helpful. Both may be ordered from Intel's Literature Department, 3065 Bowers Avenue, SV 3-3, Santa Clara, CA 95051. The supplement costs $2.00, the application note I have is not priced, but probably won't cost more than $2.00 either.

I hope you have found this material interesting and stimulating. XASM-86 and the Z-88 have given us a very low-cost development system for sophisticated 16-bit systems and software. *Puh-leeze* direct all correspondence to *Dr. Dobb's*! Our next project is underway, and there is not a chance of your letter being answered if you write here. I promise to provide XASM-86 and utilities for "the people," but someone else must pass it around. When you *do* get your hands on XASM-86, remember that the primary reason for its being was to have a development tool that would allow users to write structured, documented, maintainable code . . . that would work! ▶▶J

Late note: The Editor and I agree that keying in 17 kilobytes of object code in hex, or its equivalent of 250 pages of source code is patently ridiculous. Machine-readable code is being made available through my arrangements with RD Software, 1290 Monument St., Pacific Palisades, CA 90272. Please contact them directly.

(Listing begins on page 20)

8080/8086 Cross Assembler (Text begins on page 16)

Listing 3a.

```
**
*****************************************************************************
*
*  2 APR 81
*
*
*
* THIS PROGRAM WILL CALCULATE AND PRINT PRIME NUMBERS LESS THAN 1000.
*
* BENCHMARK (.86BENCH) REPRODUCES THE INTERFACE AGE BENCHMARK TEST FOR
* 8-BIT MICROCOMPUTERS. TIMES IN BASIC TO COMPLETE THE BENCHMARK
* RAN FROM A LOW OF 960 SECONDS (6502B) TO A HIGH OF 7950 SECONDS
* (6800B).
*
* BENCHMARK COMPLETED THE RUN IN 6 SECONDS, RUNNING ON A 4MHZ 8088.
*
* THE SPEED IMPROVEMENT RANGED FROM 160x TO 1325x THAT AVAILABLE
* FROM A HIGH LEVEL LANGUAGE RUNNING ON OTHER 8-BIT PROCESSORS.
*
* THIS PROGRAM USES A ROUTINE VERY SIMILAR TO THE DECIMAL DIGIT
* CONVERSION ROUTINE DISCUSSED IN THE TEXT OF THE DR. DOBB'S
* ARTICLE.
*
* PLEASE NOTICE THE STRUCTURE AND DOCUMENTATION OF THIS PROGRAM.
* EVEN NOVICE PROGRAMMERS WITH NO EXPERIENCE IN ASSEMBLY LANGUAGE
* HAVE BEEN ABLE TO READ AND UNDERSTAND THE OPERATION OF THIS
* PROGRAM.
*
* IN FACT, MANY PEOPLE HAVE ASSUMED THAT THIS IS A VARIANT OF
* PASCAL!!!
*




* REGISTER AND INDIRECT MEMORY REFERENCE EQUATES

TABLE_BASE:           EQUATE BX
PROGRAM_BASE:         "      AX
STACK_POINTER:        "      SP
SOURCE_POINTER:       "      BX
@_SOURCE_POINTER:     "      @BX
DESTINATION_POINTER:  "      DI
COUNTER:              "      CX
CODE_SEGMENT:         "      CS
DATA_SEGMENT:         "      DS
EXTRA_DATA_SEGMENT:   "      ES
STACK_SEGMENT:        "      SS
QUOTIENT:             "      AX
REMAINDER:            "      DX


DIVIDEND:             "      AX
```

Listing 3b.

```
**

Z80-TO-8086 CROSS-ASSEMBLER   (C) 1981 BY  JONATHAN W. MILLS

VERSION     :  1.0
VERSION DATE:  08 MAR 81
EXCEPTIONS  :  MACROS UNDER DEVELOPMENT


ADDR CODE DISP DATA  LINE LABEL:.....OPCODE....PARAMETERS.....COMMENTS.....
                     0000 *****************************************************************************
                     0010 *
                     0020 *  2 APR 81
                     0030 *
                     0040 *
                     0050 *
                     0060 * THIS PROGRAM WILL CALCULATE AND PRINT PRIME NUMBERS LESS THAN 1000.
                     0070 *
                     0080 * BENCHMARK (.86BENCH) REPRODUCES THE INTERFACE AGE BENCHMARK TEST FO
                     0090 * 8-BIT MICROCOMPUTERS. TIMES IN BASIC TO COMPLETE THE BENCHMARK
                     0100 * RAN FROM A LOW OF 960 SECONDS (6502B) TO A HIGH OF 7950 SECONDS
                     0110 * (6800B).
                     0120 *
                     0130 * BENCHMARK COMPLETED THE RUN IN 6 SECONDS, RUNNING ON A 4MHZ 8088.
                     0140 *
                     0150 * THE SPEED IMPROVEMENT RANGED FROM 160x TO 1325x THAT AVAILABLE
                     0160 * FROM A HIGH LEVEL LANGUAGE RUNNING ON OTHER 8-BIT PROCESSORS.
                     0170 *
                     0180 * THIS PROGRAM USES A ROUTINE VERY SIMILAR TO THE DECIMAL DIGIT
                     0190 * CONVERSION ROUTINE DISCUSSED IN THE TEXT OF THE DR. DOBB'S
                     0200 * ARTICLE.
                     0210 *
                     0220 * PLEASE NOTICE THE STRUCTURE AND DOCUMENTATION OF THIS PROGRAM.
                     0230 * EVEN NOVICE PROGRAMMERS WITH NO EXPERIENCE IN ASSEMBLY LANGUAGE
                     0240 * HAVE BEEN ABLE TO READ AND UNDERSTAND THE OPERATION OF THIS
                     0250 * PROGRAM.
                     0260 *
                     0270 * IN FACT, MANY PEOPLE HAVE ASSUMED THAT THIS IS A VARIANT OF
                     0280 * PASCAL!!!
                     0290 *
                     0300
                     0310
                     0320
                     0330
                     0340
                     0350 * REGISTER AND INDIRECT MEMORY REFERENCE EQUATES
                     0360
0000           1B00  0370 TABLE_BASE:       EQUATE    BX
0000           0000  0380 PROGRAM_BASE:     "         AX
0000           2400  0390 STACK_POINTER:    "         SP
0000           1B00  0400 SOURCE_POINTER:   "         BX
0000           3F00  0410 @_SOURCE_POINTE:  "         @BX
0000           3F00  0420 DESTINATION_POI:  "         DI
0000           0900  0430 COUNTER:          "         CX
0000           0900  0440 CODE_SEGMENT:     "         CS
0000           1B00  0450 DATA_SEGMENT:     "         DS
0000           0000  0460 EXTRA_DATA_SEGM:  "         ES
0000           1200  0470 STACK_SEGMENT:    "         SS
0000           0000  0480 QUOTIENT:         "         AX
0000           1200  0490 REMAINDER:        "         DX

ADDR CODE DISP DATA  LINE LABEL:.....OPCODE....PARAMETERS.....COMMENTS.....
0000           0000  0500 DIVIDEND:         "         AX
```

```
DIVISOR:              "      DI                0000          3F00  0510 DIVISOR:          "         DI
BLANK:                "      CH                0000          2D00  0520 BLANK:            "         CH
FILL_CHARACTER:       "      AL                0000          0000  0530 FILL_CHARACTER:   "         AL
PRIME:                "      AX                0000          0000  0540 PRIME:            "         AX
RESULT_POINTER:       "      BX                0000          1B00  0550 RESULT_POINTER:   "         BX
@_RESULT_POINTER:     "      @BX               0000          3F00  0560 @_RESULT_POINTE:  "         @BX
PRIME_DIGIT:          "      AL                0000          0000  0570 PRIME_DIGIT:      "         AL
N:                    "      SI                0000          3600  0580 N:                "         SI
K:                    "      BP                0000          2D00  0590 K:                "         BP
                                                                     0600
                                                                     0610
                                                                     0620
                                                                     0630
                                                                     0640
* OPERAND MACRO DEFINITIONS                                         0650 * OPERAND MACRO DEFINITIONS
                                                                     0660
none                                                                 0670
                                                                     0680
                                                                     0690
                                                                     0700
                                                                     0710
                                                                     0720
* MACRO DEFINITIONS                                                 0730 * MACRO DEFINITIONS
                                                                     0740
none                                                                 0750
                                                                     0760
                                                                     0770
                                                                     0780
                                                                     0790
                                                                     0800
* PROGRAM MODULE                                                    0810 * PROGRAM MODULE
                                                                     0820
ORIGIN at 0000H for relocatability                   0000            0830                 ORIGIN    0000H
                                                                     0840
                                                                     0850
                                                                     0860
begin the BENCHMARK: module                          0000            0870 BENCHMARK:
                                                                     0880
     this module finds all prime numbers less than .1000            0890
     it uses the interface age benchmark test from the .APR 81 IA    0900
                                                                     0910
                                                                     0920
                                                                     0930
begin routine entry initialization for a single user environment     0940
                                                                     0950
     MOVS the base address from the CODE_SEGMENT to the PROGRAM_BASE address  0000 8CC8   0960          MOVS      CODE_SEGMENT,PROGRAM_BASE
     align all other segment registers on this physical base address          0970
     MOVS the PROGRAM_BASE address to the DATA_SEGMENT               0002 8ED8   0980          MOVS      PROGRAM_BASE,DATA_SEGMENT
     MOVS the PROGRAM_BASE address to the EXTRA_DATA_SEGMENT         0004 8EC0   0990          MOVS      PROGRAM_BASE,EXTRA_DATA_SEGMENT
     MOVS the PROGRAM_BASE address to the STACK_SEGMENT              0006 8ED0   1000          MOVS      PROGRAM_BASE,STACK_SEGMENT
     LEA of the STACK into the STACK_POINTER                         0008 8D26 DA00   1010          LEA       STACK,STACK_POINTER
                                                                     1020
end the routine initialization                                       1030
                                                                     1040
                                                                     1050
                                                                     1060

                                                     ADDR CODE DISP DATA  LINE LABEL:.....OPCODE....PARAMETERS.....COMMENTS.....
                                                                     1070
                                                                     1080
begin the prime number search module                                 1090
                                                                     1100
     MVI initial value of 1 to N                     000C C7C6      0100  1110          MVI       1,N
                                                                     1120
     LOOP_N_TIMES: CPI N to 1000                     0010 81FE      E803  1130 LOOP_N_TIMES:  CPI       N,1000
          if .N is greater than 1000, then JA to EXIT 0014 77   36        1140          JA        EXIT
                                                                     1150
          else MVI initial value of 2 to K           0016 C7C5      0200  1160          MVI       2,K
```

(Continued on next page)

8080/8086 Cross Assembler (Listing continued, text begins on page 16)

Listing 3a continued.

```
      LOOP_K_TIMES: CPI K to 500
              if .K is greater than 500, then JA to PRINT_N
              this implies that .N has not been divided by any K,
              and thus is prime

              else MOV N to DIVIDEND
              XOR REMAINDER to REMAINDER to make dividend .32-bit object
              DIVide by K giving a .QUOTIENT. and a .REMAINDER.

              CPI the QUOTIENT to 0
              if .QUOTIENT = 0 this implies N = prime number
              so JE to PRINT_N

              else CPI the QUOTIENT to 1
              if .QUOTIENT = 1 this implies N not prime
              so JE to NEXT_K

              else CPI the REMAINDER to 0
              if .REMAINDER = 0, this implies that .N was evenly divided by K,
              so JE to NEXT_N

              else .N assumed not prime, try next divisor (K)

      NEXT_K: INCrement K
              JPIB to LOOP_K_TIMES

      PRINT_N: MOV the prime number N to PRIME
               LEA of RESULT to RESULT_POINTER
               CALI HEX_TO_DEC to convert .PRIME to decimal number
               CALI PRINT

      NEXT_N: INCrement N
              JPIB to LOOP_N_TIMES

  end the prime number search module




  begin the EXIT: module

      MVI the ABSOLUTE_BASE address of .0000H. to PROGRAM_BASE
      and use this to restore the entry conditions in the segment pointers.
      MOVS the PROGRAM_BASE to the DATA_SEGMENT
      MOVS the PROGRAM_BASE to the EXTRA_DATA_SEGMENT


      MOVS the PROGRAM_BASE to the STACK_SEGMENT

      MVIB 10H into AL to select the .Z80-NO NMI-NO PHANTOM-ADDR=0.
      OUTB the control word to the CPU port at FCH to avoid
      non maskable interrupt and pick up where .Z80. left off

      control now returned to .Z80. processor

  end the .EXIT module


end the .BENCHMARK module
```

```
ADDR CODE DISP DATA  LINE LABEL:.....OPCODE....PARAMETERS.....COMMENTS.....
                     1170
001A 81FD      F401  1180 LOOP_K_TIMES:  CPI     K,500
001E 77   1C         1190                JA      PRINT_N
                     1200
                     1210
                     1220
0020 89F0            1230                MOV     N,DIVIDEND
0022 31D2            1240                XOR     REMAINDER,REMAINDER
0024 F7F5            1250                DIV     K
                     1260
0026 81F8      0000  1270                CPI     QUOTIENT,0
                     1280
002A 74   10         1290                JE      PRINT_N
                     1300
002C 81F8      0100  1310                CPI     QUOTIENT,1
                     1320
0030 74   06         1330                JE      NEXT_K
                     1340
0032 81FA      0000  1350                CPI     REMAINDER,0
                     1360
0036 74   10         1370                JE      NEXT_N
                     1380
                     1390
                     1400
0038 FFC5            1410 NEXT_K:        INC     K
003A EB   DE         1420                JPIB    LOOP_K_TIMES
                     1430
003C 89F0            1440 PRINT_N:       MOV     N,PRIME
003E 8D1E DB00       1450                LEA     RESULT,RESULT_POINTER
0042 E8   1600       1460                CALI    HEX_TO_DEC
0045 E8   6700       1470                CALI    PRINT
                     1480
0048 FFC6            1490 NEXT_N:        INC     N
004A EB   C4         1500                JPIB    LOOP_N_TIMES
                     1510
                     1520
                     1530
                     1540
                     1550
                     1560
                     1570
004C                 1580 EXIT:
                     1590
004C C7C0      0000  1600                MVI     ABSOLUTE_BASE,PROGRAM_BASE
                     1610
0050 8ED8            1620                MOVS    PROGRAM_BASE,DATA_SEGMENT
0052 8EC0            1630                MOVS    PROGRAM_BASE,EXTRA_DATA_SEGMENT

ADDR CODE DISP DATA  LINE LABEL:.....OPCODE....PARAMETERS.....COMMENTS.....
0054 8ED0            1640                MOVS    PROGRAM_BASE,STACK_SEGMENT
                     1650
0056 C6C0      10    1660                MVIB    10H,AL
0059 E6        FC    1670                OUTB    FCH
                     1680
                     1690
                     1700
                     1710
                     1720
                     1730
                     1740
                     1750
                     1760
                     1770
                     1780
                     1790
```

Listing 3b continued.

```
                                                                    1800
                                                                    1810
begin the HEX_TO_DEC: module                        005B            1820 HEX_TO_DEC:
                                                                    1830
                                                                    1840
                                                                    1850
            MVIB ' ' to BLANK to blank leading zeroes               1860                MVIB    ' ',BLANK
            MOVB BLANK to leftmost byte @_RESULT_POINTER            1870                MOVB    BLANK,@_RESULT_POINTER
            MVI 10000 to DIVISOR                    0060 C7C7  1027 1880                MVI     10000,DIVISOR
            CMP PRIME to DIVISOR                    0064 3BC7       1890                CMP     PRIME,DIVISOR
            if .PRIME is less than DIVISOR, then JB to THOUSANDS    1900                JB      THOUSANDS
            else CALI DIGIT to make ten thousands digit             1910                CALI    DIGIT
                                                                    1920
THOUSANDS:  INCrement RESULT_POINTER                006B FFC3       1930 THOUSANDS:     INC     RESULT_POINTER
            MOVB BLANK to next byte @_RESULT_POINTER    006D 882F   1940                MOVB    BLANK,@_RESULT_POINTER
            MVI 1000 to DIVISOR                     006F C7C7  E803 1950                MVI     1000,DIVISOR
            CMP PRIME to DIVISOR                    0073 3BC7       1960                CMP     PRIME,DIVISOR
            if .PRIME is less than DIVISOR, then JB to HUNDREDS     1970                JB      HUNDREDS
            else CALI DIGIT to make thousands digit 0077 E8   2600  1980                CALI    DIGIT
                                                                    1990
HUNDREDS:   INCrement RESULT_POINTER                007A FFC3       2000 HUNDREDS:      INC     RESULT_POINTER
            MOVB BLANK to next byte @_RESULT_POINTER    007C 882F   2010                MOVB    BLANK,@_RESULT_POINTER
            MVI 100 to DIVISOR                      007E C7C7  6400 2020                MVI     100,DIVISOR
            CMP PRIME to DIVISOR                    0082 3BC7       2030                CMP     PRIME,DIVISOR
            if .PRIME is less than DIVISOR, then JB to TENS         2040                JB      TENS
            else CALI DIGIT to make hundreds digit  0086 E8   1700  2050                CALI    DIGIT
                                                                    2060
TENS:       INCrement RESULT_POINTER                0089 FFC3       2070 TENS:          INC     RESULT_POINTER
            MOVB BLANK to next byte @_RESULT_POINTER    008B 882F   2080                MOVB    BLANK,@_RESULT_POINTER
            MVI 10 to DIVISOR                       008D C7C7  0A00 2090                MVI     10,DIVISOR
            CMP PRIME to DIVISOR                    0091 3BC7       2100                CMP     PRIME,DIVISOR
            if .PRIME is less than DIVISOR, then JB to ONES         2110                JB      ONES
            else CALI DIGIT to make tens digit      0095 E8   0800  2120                CALI    DIGIT
                                                                    2130
ONES:       INCrement RESULT_POINTER                0098 FFC3       2140 ONES:          INC     RESULT_POINTER
            ADIB ASCII_CONSTANT to PRIME_DIGIT      009A 80C0   30  2150                ADIB    ASCII_CONSTANT,PRIME_DIGIT
            MOVB PRIME_DIGIT to byte @_RESULT_POINTER   009D 8807   2160                MOVB    PRIME_DIGIT,@_RESULT_POINTER
            RETurn                                  009F C3         2170                RET
                                                                    2180
                                                                    2190
                                                                    2200

                                                    ADDR CODE DISP DATA  LINE LABEL:.....OPCODE....PARAMETERS.....COMMENTS.....
end the .HEX_TO_DEC module                                          2210
                                                                    2220
                                                                    2230
                                                                    2240
                                                                    2250
                                                                    2260
begin the DIGIT: module                             00A0            2270 DIGIT:
                                                                    2280
                                                                    2290
                                                                    2300
    XOR REMAINDER to REMAINDER to zero hi word of .32-bit PRIME 00A0 31D2   2310                XOR     REMAINDER,REMAINDER
    DIVide .PRIME. by the DIVISOR                   00A2 F7F7       2320                DIV     DIVISOR
    ADIB ASCII_CONSTANT to PRIME_DIGIT              00A4 80C0   30  2330                ADIB    ASCII_CONSTANT,PRIME_DIGIT
    MOVB PRIME_DIGIT to byte @_RESULT_POINTER       00A7 8807       2340                MOVB    PRIME_DIGIT,@_RESULT_POINTER
    MOV REMAINDER to PRIME                          00A9 89D0       2350                MOV     REMAINDER,PRIME
    MVIB ASCII_CONSTANT to BLANK to place zeroes into the   00AB C6C5   30  2360          MVIB    ASCII_CONSTANT,BLANK
    result instead of blanks after a digit has been inserted    2370
    RETurn                                          00AE C3         2380                RET
                                                                    2390
                                                                    2400
                                                                    2410
end the .DIGIT module                                               2420
                                                                    2430
                                                                    2440
                                                                    2450
                                                                    2460
```

(Continued on next page)

8080/8086 Cross Assembler (Listing continued, text begins on page 16)

Listing 3a continued.

```
begin the PRINT: module



     LEA of the RESULT into the SOURCE_POINTER
     MVI the PRINT_LENGTH into the COUNTER

begin the PRINT_LOOP:

     get the status INB 0
     ANIB 80H with the status in AL
     if not ready to send character then JZ to PRINT_LOOP

     MOVB a character from @_SOURCE_POINTER to PRIME_DIGIT
     send the .PRIME_DIGIT. OUTB data port 1
     INCrement the SOURCE_POINTER
     LOOP back to PRINT_LOOP until the count is zero

end the  PRINT_LOOP

     RETurn to the calling program



end the .PRINT module
```

```
* OPERAND DEFINITIONS

NYBBLE:         EQUATE FH
ABSOLUTE_BASE:   " 0
PRINT_LENGTH:    " 7
ASCII_CONSTANT: " 30H
```

```
* DATA DEFINITIONS

 ALLOCATE 20 bytes for a stack
 since the stack grows toward low memory locate the
 STACK: at high memory to start by EQUATE to STACK-1
```

```
RESULT: in ascii of a hexadecimal word
 ALLOCATE 5 characters in the result
 end each result with BYTE 0DH carriage return
 and " 0AH linefeed
```

```
END of the source listing
```

End Listing 3a.

Listing 3b continued.

```
00AF                    2470
00AF                    2480 PRINT:
                        2490
                        2500
                        2510
00AF 8D1E DB00          2520                LEA     RESULT,SOURCE_POINTER
00B3 C7C1      0700     2530                MVI     PRINT_LENGTH,COUNTER
                        2540
00B7                    2550 PRINT_LOOP:
                        2560
00B7 E4          00     2570                INB     0
00B9 80E0        80     2580                ANIB    80H,AL
00BC 74   F9            2590                JZ      PRINT_LOOP
                        2600
00BE 8A07               2610                MOVB    @_SOURCE_POINTER,PRIME_DIGIT
00C0 E6          01     2620                OUTB    1
00C2 FFC3               2630                INC     SOURCE_POINTER
00C4 E2   F1            2640                LOOP    PRINT_LOOP
                        2650
                        2660
                        2670
00C6 C3                 2680                RET
                        2690
                        2700
                        2710
                        2720
                        2730
                        2740
                        2750
                        2760
                        2770
```

```
ADDR CODE DISP DATA  LINE LABEL:.....OPCODE....PARAMETERS.....COMMENTS.....
                     2780 * OPERAND DEFINITIONS
                     2790
00C7           0F00  2800 NYBBLE:          EQUATE    FH
00C7           0000  2810 ABSOLUTE_BASE:   "         0
00C7           0700  2820 PRINT_LENGTH:    "         7
00C7           3000  2830 ASCII_CONSTANT:  "         30H
                     2840
                     2850
                     2860
                     2870
                     2880
                     2890 * DATA DEFINITIONS
                     2900
00C7                 2910                   ALLOCATE  20
                     2920
00DB           DA00  2930 STACK:            EQUATE    STACK-1
                     2940
                     2950
                     2960
                     2970
                     2980
00DB                 2990 RESULT:
00DB                 3000                   ALLOCATE  5
00E0           0D    3010                   BYTE      0DH
00E1           0A    3020                   "         0AH
                     3030
                     3040
                     3050
                     3060
                     3070
00E2                 3080
```

End Listing 3b.

Small-C DISKDOC

A Repair and Maintenance Utility

by Egil Kvaleberg

Egil Kvaleberg, Studpost 111, N-7034 Trondheim-NTH, Norway.

Diskdoc is a utility which, like its name implies, is intended for repair and maintenance of diskettes and other random access mass storage media. Tasks accomplished include:

- Making backups (also in a single drive system).
- Verifying data integrity.
- Restoring previously deleted files (by manual patching).
- Repairing faulty sectors.
- Debugging disk driver routines and controller hardware.
- Debugging file systems.

Diskdoc has been in frequent use for about two years now (including a one-year assembly language childhood), and has proved a valuable companion when exploring the oh-so-vulnerable world of diskettes.

The version shown here runs under CP/M, but adapting to other operating systems will hopefully not be found too difficult a task to accomplish. As explained in the listing, the program is written to be compiled by the Ron Cain Small-C compiler. Small-C *is* smaller than standard C, and connoisseurs of the C language will find some of the constructions used quite primitive. If you prepare the program for a richer dialect of C, you might consider rewriting parts of the program. Except for efficiency, perhaps, there is little reason to take the bother since Small-C is a true subset of C.

Talking To Diskdoc

Diskdoc always tries to give self-explanatory prompts. The default value can usuallly be displayed by typing <cr> once. If you should enter an inappropriate character, Diskdoc simply refuses to echo it. Entries may be corrected by using the backspace key. A final <cr> will, as usual, end the input line.

The Command Level

Starting up, Diskdoc will enter the command level, requesting a single-letter command:

a Adapt to terminal type. You will be asked to enter the character that will move the cursor home. An excape sequence is acceptable. This is all that is required to adapt Diskdoc to virtually any video terminal. It is assumed that the width of the screen is 80 characters.

b Backup entire diskette, on a sector-by-sector basis. You will be asked to enter the source and destination drive names (a-h). It is crucial to double check the direction of data flow. The same drive might be assigned as both source and destination, in which case Diskdoc will request a diskette change whenever necessary during the copy process. If an error is detected in the source diskette, Diskdoc will ask whether the sector read operation should be retried, the error ignored, or the backup aborted. It is recommended to retry a few times before declaring a sector unrecoverable, thereby filtering out intermittently occuring errors. If desired, backup can be aborted by typing any character.

c Compare the contents of two diskettes. Every discrepancy will be reported. Abort by typing any character.

e Exit. You will be prompted to insert a system diskette before leaving.

p Patch. Look at and optionally patch selected sectors. For patch subcommands, see below.

s Scan diskette by reading every sector. The test will quickly locate any sector that doesn't give a proper CRC. The test may be interrupted by typing any character.

t Test diskette. A test pattern will be written, and the entire diskette will then be read back and compared. The write operation may be omitted. The command provides useful verification of disk controller hardware and software operation.

z Zero diskette. The specified number of tracks will be erased by filling the EBCDIC character "V" (0e5h).

Patch Commands

<sp> Move the cursor to the next byte.

<bs> Move the cursor to the previous byte.

<cr> Move the cursor down, i.e., 16 bytes forward.

<hex> To patch, enter the new hex byte directly. After having entered the first digit, a <bs> will "undo" it. After having entered the second, the cursor will move to the next byte.

' To enter an ASCII character, type a single quote followed by the desired character.

l Read the logically following sector. The command is useful where a mapping between physical and logical sectors exists, such as in standard CP/M single-density diskettes.

n Read the next sequential sector. The next track is accessed when the end of the current one is reached.

q Quit patch mode.

r Re-read. Read the current sector once again.

s Shift the rest of the sector one bit-position to the right. The command is useful when attempting to repair sectors where the disk controller has lost a few bits here and there.

t Select a new track and/or sector.

w Write back the current sector. Remember to use this command after having made any changes, since no automatic write is performed.

Regarding the CP/M Version

The CP/M version, as listed, will adapt to the physical characteristics of the drive being used (track and sector counts). This is accomplished by reading the disk parameter block returned by the BIOS drive select call. CP/M version 2.0 or better is required. For drives using sector blocking/deblocking schemes, the sectors that Diskdoc works with will not correspond directly to the physical sectors.

Adapting to Other Operating Systems and/or Compilers

All implementation-dependent functions are contained in the file "ddocsys.c". These functions are all very low-level, and thus very non-standard. You will have to write these functions yourself, since no C runtime library will support them. If automatic size adaptation and/or logical to physical sector mapping aren't required, the functions seldrv() and lnaext(), respectively can be considerably simplified.

(Listing begins on page 28)

Small-C DISKDOC (Text begins on page 26)

See bug fixes, DDJ Bound Volume page 193.

```
/*  TAB S1,4            FILE DISKDOC.C
 *
 *  diskdoc
 *  a utility for diskette maintainance
 *
 *  by egil kvaleberg
 *     studpost 111
 *     n7034 trondheim-nth
 *     norway
 *
 *  diskdoc is intended for use and distribution
 *  among amateur computer users only. please respect this.
 *
 *  compile using the Ron Cain small C compiler as distributed
 *  by The Code Works. if you use the version of the compiler shown
 *  in the Dr. Dobbs May 1980 issue, you must ensure that the stack
 *  is properly initialized. if you use another C compiler, you'll
 *  have to revise (parts of) the file ddocsys.c. good luck.
 */

#define VERSION "ver c - 1 nov 81"
#define WHOMADEIT "by egil kvaleberg"

/*
 *  constants, change as required
 */

#define SECLEN 128      /* sector length, 128 or 256 is ok */

#define VOID 229        /* character used when zeroing diskette */

#define DHOME 26        /* default home character */
#define ESC 27          /* lead-in character */

#define CR 13           /* various ascii equates */
#define LF 10
#define BS 8
#define QOT 39

/*
 *  globals
 */

int tracks,             /* number of tracks and sectors */
    sectors,firstsector;

int dfltrk,dflsec;      /* default values */
char dfldrv;

char secbuf[SECLEN];    /* buffer for patch */
char vfybuf[SECLEN];    /* buffer for write verify */

int home;               /* character to move cursor home */

/*
 *  begin
 */

main()
{   char cmd;
    dfldrv='a'; dfltrk=0; dflsec=0;
    home=DHOME;
    if (sysok()==0) exit();
    nl(); puts("welcome to the world of diskdoc ");
    nl(); puts(VERSION); nl(); puts(WHOMADEIT);
    nl(); nl(); puts("kindly enter your request");
    cmd=0;
    while (cmd!='e') {

        nl();
        puts("adapt, backup, compare, exit, patch, scan, test or zero ?");
        cmd=getcmd("abcepstz",'e');
        if (cmd=='a') adapt();
        else if (cmd=='b') backup();
        else if (cmd=='c') compare();
        else if (cmd=='p') patch();
        else if (cmd=='s') scan();
        else if (cmd=='t') test();
        else if (cmd=='z') zero();
        else {
            nl(); puts("re-insert the system diskette and type <cr> ");
            if (conlower()!=CR) cmd=0;
            nl();
        }
    }
    nl(); puts("thanks for having consulted diskdoc");
    exit();
}

/*
 *  adapt to terminal
 */

adapt()
{   nl(); nl(); puts("type character to move cursor home..");
    putbyte(home=conin());
    if (home==ESC) {
        puts(".."); putbyte(home=conin());
        home=home+128;
    }
    nl();
}

/*
 *  backup, copy entire diskette
 */

backup()
{   int trk,ftrk,ltrk;
    char sdrive,ddrive; char *adr;
    if (seltwo(&sdrive,&ddrive,
              "source diskette in drive (a-h) ?",
              "destination diskette in drive (a-h) ?")==0) return;
    if (cont()==0) return;
    trk=-1;
    while (nextt(&trk)) {
        adr=botmem();
        ftrk=trk;
        if (baksel(sdrive,ddrive,"source")==0) return;
        while (1) {
            if (readtrk(trk,adr)==0) return;
            ltrk=trk;
            adr=adr+(SECLEN*sectors);
            if ((adr+(SECLEN*sectors))>=topmem()) break;
            if (nextt(&trk)==0) break;
        }
        adr=botmem(); trk=ftrk;
        if (baksel(ddrive,sdrive,"destination")==0) return;
        while (1) {
            if (writetrk(trk,adr)==0) {
                puts("backup aborted"); nl();
                return;
            }
            if (trk>=ltrk) break;
            nextt(&trk);
            adr=adr+(SECLEN*sectors);
        }
    }
    nl(); puts("backup finished"); nl();
}
/*
 *  compare contents of two diskettes
 */

compare()
{   int trk,sec;
    char drive1,drive2;
    char *adr1,*adr2;
    if (seltwo(&drive1,&drive2,
              "compare diskette in drive (a-h) ?",
              "to diskette in drive (a-h) ?")==0) return;
    if (cont()==0) return;
    trk=-1;
    while (nextt(&trk)) {
        adr1=botmem();
        adr2=adr1+(SECLEN*sectors);
        if (baksel(drive1,drive2,"first")==0) return;
        if (readtrk(trk,adr1)==0) return;
        if (baksel(drive2,drive1,"second")==0) return;
        if (readtrk(trk,adr2)==0) return;
        sec=firstsector-1;
        while (nexts(&sec)) {
            if (eqsec(adr1,adr2)==0) {
                dfltrk=trk; dflsec=qskew(sec);
                nl(); puts("compare error in "); putsec(dfltrk,dflsec);
            }
            adr1=adr1+SECLEN; adr2=adr2+SECLEN;
        }
    }
    nl(); puts("compare finished"); nl();
}

/*
 *  zero diskette contents
 */

zero()
{   int trk,trkmax,n,m;
    char *adr;
    nl(); nl(); puts("zero diskette in drive (a-h) ?");
    if (getsel()==0) return;
    nl(); puts("zero how many tracks (0-"); putnum(tracks-1); puts(") ?");
    trkmax=getnum(tracks-1,0,tracks-1);
    if (cont()==0) return;
    n=0; adr=botmem(); m=SECLEN*sectors;
    while (n<m) adr[n++]=VOID;
    trk=0;
    while (trk<trkmax) {
        if (writetrk(trk,botmem())==0) return;
        nextt(&trk);
    }
    nl();
}

/*
 *  test a diskette by writing a pattern first,
 *  and then reading it back
 */

test()
{   int trk,sec,n;
    char *adr,*p;
    nl(); nl(); puts("diskette to test in drive (a-h) ?");
    if (getsel()==0) return;
```

```
    nl(); puts("the test will destroy the contents of the diskette");
    if (cont()==0) return;
    p="**** diskdoc **** test pattern *"; /* length is 32 */
    n=0;            /* fill pattern to compare to */
    while (n<SECLEN) {
        secbuf[n]=p[n&31];
        ++n;
    }
    puts("does diskette already contain test pattern (y-n) ?");
    if (getcmd("yn",'n')=='n') {
        nl(); puts("writing test pattern...");
        trk=0; sec=firstsector-1;
        while (next(&trk,&sec)) {
            secbuf[0]=trk; secbuf[1]=qskew(sec);
            if (tstbrk()) return 0;
            if (write(trk,qskew(sec),secbuf)==0) {
                nl(); puts("error writing "); putsec(trk,qskew(sec)); nl();
                return 0;
            }
        }
    }
    trk=-1;         /* read test */
    nl(); puts("reading test pattern...");
    while (nextt(&trk)) {
        if (readtrk(trk,botmem())==0) return;
        sec=firstsector-1; adr=botmem();
        while (nexts(&sec)) {
            secbuf[0]=trk; secbuf[1]=qskew(sec);
            if (eqsec(adr,secbuf)==0) {
                dfltrk=trk; dflsec=qskew(sec);
                nl(); puts("bad test pattern in "); putsec(dfltrk,dflsec);
            }
            adr=adr+SECLEN;
        }
    }
    nl(); puts("test completed"); nl();
}

/*
 *  scan: check all sectors for crc errors etc.
 */

scan()
{   int trk,sec;
    nl(); nl(); puts("scan diskette in drive (a-h) ?");
    if (getsel()==0) return;
    if (cont()) {
        nl();
        trk=0; sec=firstsector-1;
        while (next(&trk,&sec)) {
            if (tstbrk()) return;
            if (sec==firstsector) {
                puts("track "); putnum(trk); conout(CR);
            }
            if (read(trk,qskew(sec),vfybuf)==0) {
                puts("the dubious quality of ");
                putsec(trk,qskew(sec)); puts(" has been detected"); nl();
                dfltrk=trk; dflsec=qskew(sec);
            }
        }
        puts("scan finished"); nl();
    }
}

/*
 *  patch
 */

patch()
{   int pos;
    nl(); nl(); puts("patch diskette in drive (a-h) ?");
    if (getsel()==0) return;
    if (dflsec<firstsector) dflsec=firstsector;
    getsec(&dfltrk,&dflsec);
    putnls(60); gohome();
    puts("<sp>=next-byte, <bs>=previous-byte, <cr>=next-line, <hex>, ',");
    nl();
    puts("next, logical-next, trk-sec, re-read, shift-bit, write or quit ?");
    putnls(4);
    puts("    +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +a +b +c +d +e +f");
    puts("   0123456789abcdef");
    pos=pread();
    while ((pos=patchcmd(pos))>=0);
    putnls(60);
}

/*
 *  one patch command
 *  return new position, -1 if quit
 */

patchcmd(pos)
    int pos;
{   char cmd,dig;
    cmd=conlower();
    if (cmd=='q') return -1;
    else if (cmd==' ') pos=(pos+1)%SECLEN;
    else if (cmd==BS)  pos=(pos+SECLEN-1)%SECLEN;
    else if (cmd==CR)  pos=(pos+16)%SECLEN;
    else if (cmd=='s') pos=pshift(pos);
    else if (cmd=='n') pos=pnsect();
    else if (cmd=='l') pos=plsect();
    else if (cmd=='t') pos=psetts();
    else if (cmd=='r') pos=pread();
    else if (cmd=='w') pos=pwrite();
    else if (cmd==QOT) {
        conout(QOT);
        cmd=conin();
        conout(BS);
        if (cmd==BS) tbyte(pos);
        else pos=patchpos(pos,cmd);
    }
    else if ((dig=makhex(cmd))>=0) {
        putdig(dig);
        while ((cmd=conlower())!=BS) if (makhex(cmd)>=0) break;
        conout(BS);
        if (cmd==BS) tbyte(pos);
        else pos=patchpos(pos,(dig<<4)+makhex(cmd));
    }
    return poscur(pos);
}

/*
 *  patch one byte, update screen
 *  and return new position
 */

patchpos(pos,new)
    int pos,new;
{   int tpos;
    secbuf[pos]=new;
    tpos=pos++;
    if (pos>=SECLEN) pos=0;
    while (tbyte(tpos++)>0);
    return pos;
}

/*
 *  shift one bit at current position
 */

pshift(pos)
    int pos;
{   int tpos,byte,carry;
    tpos=pos;
    carry=0;
    while (1) {
        byte=secbuf[tpos]&255;
        secbuf[tpos]=(byte>>1)|(carry<<7);
        carry=byte&1;
        if (tbyte(tpos++)==0) break;
    }

    return poscur(pos);
}

/*
 *  next sector
 */

pnsect()
{   next(&dfltrk,&dflsec);
    return pread();
}

/*
 *  next logical sector
 *  useful only in systems with mapping between logical and physical
 *  sector numbers
 */

plsect()
{   lnext(&dfltrk,&dflsec);
    return pread();
}

/*
 *  set new track/sector address
 */

psetts()
{   gohome(); putnls(2); putsps(38);
    gohome(); nl();
    getsec(&dfltrk,&dflsec);
    return pread();
}

/*
 *  read and display one sector during patch
 */

pread()
{   char *p;
    int pos;
    pos=0;
    while (pos<SECLEN) secbuf[pos++]=0;
    gohome(); putnls(2);
    if (read(dfltrk,dflsec,secbuf)==0) puts("error reading ");
    putsec(dfltrk,dflsec);
    putsps(18); nl(); putsps(24);
    poscur(pos=0);
    while (tbyte(pos++));
    return poscur(0);
}

/*
 *  write one sector during patch
 *  verify by reading back
 */

pwrite()
{   int c;
    gohome(); putnls(2); c=cont();
```

(Continued on next page)

```
    gohome(); putnls(3);
    if (c) {
        if (write(dfltrk,dflsec,secbuf)==0) puts("error writing");
        else if (read(dfltrk,dflsec,vfybuf)==0) puts("error reading back");
        else if (eqsec(secbuf,vfybuf)==0) puts("error on read verify");
    }
    putsps(22);
    return poscur(0);
}

/*
 *  position cursor during patch
 *  position is returned
 */

poscur(pos)
    int pos;
{   int n;
    gohome(); putnls(6); puts("00  ");
    n=pos/16; while (n--) conout(LF);
    n=(pos/16)*16; while (n<pos) tbyte(n++);
    return pos;
}

/*
 *  print byte in patch table
 *  returns -1 if end of line
 *          0 if end of page
 *          1 otherwise
 */

tbyte(pos)
    int pos;
{   putbyte(secbuf[pos++]); /* print byte */
    puts(" ");
    if (pos%16) return 1;
    puts(" '");             /* print ascii equivalents */
    pos=pos-16;
    while (1) {
        conout(makep(secbuf[pos++]));
        if ((pos%16)==0) break;
    }
    puts("'"); nl();
    if (pos==SECLEN) return 0;
    putbyte(pos);           /* print address too */
    puts("  ");
    return -1;
}

/*
 *  read one track
 *  assume that drive is already selected
 */

readtrk(trk,adr)
    int trk;
    char *adr;
{   int sec;
    char cmd;
    sec=firstsector;
    while (1) {
        if (tstbrk()) return 0;
        if (read(trk,qskew(sec),adr)==0) {
            nl(); puts("error reading "); putsec(trk,qskew(sec));
            nl(); puts("continue, retry or quit ?");
            cmd=getcmd("qcr",'r');
            if (cmd=='r') {
                read(0,firstsector,vfybuf); /* reposition */
                continue;
            }
            dfltrk=trk; dflsec=qskew(sec);
            if (cmd=='q') return 0;
        }
        adr=adr+SECLEN;
        if (nexts(&sec)==0) return 1;
    }
}

/*
 *  write one track
 */

writetrk(trk,adr)
    int trk;

    char *adr;
{   int sec;
    sec=firstsector;
    while (1) {
        if (tstbrk()) return 0;
        if (write(trk,qskew(sec),adr)==0) {
            nl(); puts("error writing "); putsec(trk,qskew(sec)); nl();
            return 0;
        }
        adr=adr+SECLEN;
        if (nexts(&sec)==0) return 1;
    }
}

/*
 *  get track and sector numbers
 */

getsec(t,s)
    int *t,*s;
{   nl(); puts("track (0-"); putnum(tracks-1); puts(") ?");
    *t=getnum(*t,0,tracks-1);
    nl(); puts("sector ("); putnum(firstsector); puts("-");
    putnum(sectors-1+firstsector); puts(") ?");
    *s=getnum(*s,firstsector,sectors-1+firstsector);
}

/*
 *  get drive name, select and reset
 *  0 if cannot select
 */

getsel()
{   if (seldrv((dfldrv=getdrv(dfldrv)),&tracks,&sectors,&firstsector)==0) {
        nl(); puts("drive not ready"); nl();
        return 0;
    }
    rstdrv();
    return 1;
}

/*
 *  select two drives
 *  parameters of drives selected must be equivalent
 */

seltwo(d1,d2,text1,text2)
    char *d1,*d2;
    char *text1,*text2;
{   int dtracks,dsectors,dfirstsector;
    nl(); nl(); puts(text1);
    *d1=dfldrv=getdrv(dfldrv);
    nl(); puts(text2);
    *d2=getdrv(dfldrv);
    seldrv(*d1,&tracks,&sectors,&firstsector);      /* establish constants */
    rstdrv();
    seldrv(*d2,&dtracks,&dsectors,&dfirstsector);
    rstdrv();
    if ((dtracks!=tracks)|(dsectors!=sectors)|(dfirstsector!=firstsector)) {
        nl(); puts("drives are not compatible"); nl();
        return 0;
    }
    return 1;
}

/*
 *  increment track and sector, 0 if no more
 */

next(t,s)
    int *t,*s;
{   if (nexts(s)) return 1;


    return nextt(t);
}

/*
 *  increment sector, 0 if no more
 */

nexts(s)
    int *s;
{   if ((++*s)>(sectors-1+firstsector)) {
        *s=firstsector;
        return 0;
    }
    return 1;
}

/*
 *  increment track, 0 if no more
 */

nextt(t)
    int *t;
{   if ((++*t)>=tracks) return *t=0;
    return 1;
}

/*
 *  return skewed sector number, skew factor is two.
 *  this function is not equivalent to lnext().
 *  the only purpose of this function is to optimize
 *  scan and backup operations.
 *  hard disks will operate faster without a skew.
 *  hard disk is assumed if there are more than 100 tracks (!).
 */

qskew(sec)
    int sec;
{   if (tracks>100) return sec;
    sec=sec+sec-firstsector;
    if (sec>(sectors-1+firstsector))
        return sec-sectors+((sectors&1)==0);
    return sec;
}
```

```
/*
 *  compare two sectors
 *  false if not equal
 */

eqsec(sec1,sec2)
    char *sec1,*sec2;
{   int n;
    n=0;
    while (n<SECLEN) {
        if (sec1[n]!=sec2[n]) return 0;
        ++n;
    }
    return 1;
}

/*
 *  select drive during backup
 *  return 0 if something fishy
 */

baksel(drv,otherdrv,name)
    char drv,otherdrv; char *name;
{   int dummy;
    if (drv==otherdrv) {
        nl(); puts("insert "); puts(name); puts(" diskette");
        if (cont()==0) return 0;
    }

    if (seldrv(drv,&dummy,&dummy,&dummy)==0) {
        nl(); puts(name); puts(" drive is not ready"); nl();
        return 0;
    }
    return 1;
}

/*
 *  get drive name, enter with default drive
 */

getdrv(dfl)
    char dfl;
{   return getcmd("abcdefgh",dfl);
}

/*
 *  ask for continue or quit, true if continue
 */

cont()
{   char cmd;
    nl(); puts("continue or quit ?");
    cmd=getcmd("cq",'c'); nl();
    return cmd=='c';
}

/*
 *  get single character command,
 *  enter with possible commands and default command
 */

getcmd(cmds,dfl)
    char *cmds,dfl;
{   char cmd,ch;
    while (1) {
        if ((cmd=conlower())==CR) cmd=dfl;
        if (member(cmd,cmds)==0) continue;
        conout(cmd);
        ch=0;
        while (ch!=BS) if ((ch=conlower())==CR) return cmd;
        unput();
    }
}

/*
 *  see if character can be found in a string
 */

member(ch,str)
    char ch,*str;
{   while (*str) if (*str++==ch) return 1;
    return 0;
}

/*
 *  get a unsigned decimal number
 *  enter with default, minimun and maximum values
 */

getnum(dfl,min,max)
    int dfl,min,max;
{   int n,digits,newn; char ch;
    if (dfl>max) dfl=max;
    n=digits=0;
    while (1) {
        if ((ch=conlower())==CR) {
            if (digits) {
                if (n>=min) break;
            } else digits=putnum(n=dfl);
        } else if (ch==BS) {
            if (digits) {

                --digits; n=n/10;
                unput();
            }
        } else if (((makdec(ch))>=0) & ((newn=n*10+makdec(ch))<=max)) {
            if ((newn>0)|(digits==0)) {
                ++digits; n=newn;
                conout(ch);
            }
        }
    }
    return n;
}

/*
 *  make character printing
 */

makep(ch)
    char ch;
{   ch=ch&127;
    if ((ch>=' ')&(ch<127)) return ch;
    return ' ';
}

/*
 *  put track and sector numbers
 */

putsec(t,s)
    int t,s;
{   puts("track "); putnum(t);
    puts(" sector "); putnum(s);
}

/*
 *  put decimal number,
 *  return number of digits printed
 */

putnum(n)
    int n;
{   int d;
    if (n>9) d=putnum(n/10);
    else d=0;
    putdig(n%10);
    return d+1;
}

/*
 *  put hex byte
 */


putbyte(n)
    int n;
{   putdig(n>>4);
    putdig(n);
}

/*
 *  put digit
 */

putdig(d)
    int d;
{   if ((d=d&15)>9) conout(d-10+'a');
    else conout(d+'0');
}

/*
 *  put string
 */

puts(str)
    char *str;
{   while (*str) conout(*str++);
}

/*
 *  put spaces
 */

putsps(n)
    int n;
{   while (n--) conout(' ');
}

/*
 *  new line
 */

nl()
{   conout(CR); conout(LF);
}

/*
 *  put newlines
 */

putnls(n)
    int n;
{   while (n--) nl();
}

/*
 *  erase previous character
 */

unput()
{   conout(BS); conout(' '); conout(BS);
}

/*
 *  move cursor to home position
 */
```

```
gohome()
{   if (home>=128) conout(ESC);
    conout(home & 127);
}

/*
 *  convert decimal digit, -1 if error
 */

makdec(ch)
    char ch;
{   if ((ch=ch-'0')<0) return -1;
    if (ch<=9) return ch;
    return -1;
}

/*
 *  convert hex digit, -1 if error
 */

makhex(ch)
    char ch;
{   if (ch<'0') return -1;
    if (ch<='9') return ch-'0';
    if ((ch=ch-'a'+10)<10) return -1;
    if (ch<=15) return ch;
    return -1;
}

/*
 *  test for break, true if yes
 */

tstbrk()
{   if (const()) {
        nl(); puts("break");
        if (cont()==0) return 1;
    }
    return 0;
}

/*
 *  get character, no echo
 *  upper case is converted to lower
 */


conlower()
{   char ch;
    if (((ch=conin())>='A')&(ch<='Z')) ch=ch+'a'-'A';
    return ch;
}



/*
 *  system dependant functions
 */

#include ddocsys.c



/*
 *  the small C runtime library (as shown in the Dr. Dobbs
 *  September 1980 issue and as distributed by The Code
 *  Works) must be named ddoclib and edited
 *  so that it contains the primitive functions only:
 *  ccgchar,ccgint,ccpchar,...,ccmult,ccdiv
 */

#include ddoclib



 /*  TAB S1,4            FILE DDOCSYS.C
  *
  *  system dependant functions for diskdoc
  *
  *  required are:
  *  botmem,topmem,sysok,conin,conout,const
  *  seldrv,lnext,read,write,rstdrv
  */

 /*
  *  this version is for small C and cp/m 8080 version 2.x.
  *  since bios must be used for sector read and write,
  *  character i/o functions use bios directly too.
  *  cp/m "compatible" operating systems will probably
  *  need to have these functions rewritten.
  */

 char *xlt;      /* sector xlate table, used by cp/m version */

 /*
  *  return pointer to bottom of free memory
  */

 int lastglobal;     /* dirty trick, will work in small-c */

 botmem()
 {   return &lastglobal;
 }

 /*
  *  top of free memory
  *  remember that the stack needs some space too
  */

 topmem()
 {   char *p,topofstack;
     p=&topofstack;
     return p-300;
 }

 /*
  *  return true if enviroment seems to be ok
  */

 sysok()
 {   char *p;
 #asm
     mvi     c,12        ;check version number
     call    5
     ani     0f0h
     cpi     20h         ;ver 2.x?
     jnz     nogood

     mov     a,h
     ora     a           ;cp/m?
     jz      ok
 nogood:
     lxi     h,0         ;false if no good
     pop     d
     ret
 ok:
 #endasm
     p=1;
     return (*p==3);     /* check if xsub or despool is active */
 }



 /*
  *  return console status, true if ready
  */

 const()
 {   return bios(2,0);
 }

 /*
  *  get character from console, no echo
  */

 conin()
 {   return bios(3,0);
 }

 /*
  *  put character to console
  *  no character conversion should be performed
  */

 conout(ch)
     char ch;
 {   bios(4,ch);
 }



/*
 *  select drive, drive name is 'a','b' etc.
 *  set values for track, sector counts and first sector
 *  return true if ok
 *  to see what this version does,
 *  refer to the cp/m 2.0 alteration guide
 */



seldrv(drv,pt,ps,pf)
    char drv;
    int *pt,*ps,*pf;        /* where to put track, sector and firstsector */
{   int *dph,*spt,*dsm,*off,halfsecs,trks;
    char *dpb,*bls;
    if ((dph=bios(9,drv-'a'))==0) return 0;
    xlt=dph[0];             /* look at disk parameter header */
    dpb=dph[5];
    if (xlt) *pf=xlt[0];
    else *pf=0;
    spt=&dpb[0]; *ps=*spt;  /* and at disk parameter block too */
```

```
    bls=&dpb[2];
    dsm=&dpb[5];
    off=&dpb[13];
    /* this is tricky since unsigned divide isn't supported */
    halfsecs=(*dsm+1)<<(*bls-1);
    trks=((halfsecs/(*spt))*2)+(((halfsecs%(*spt))+(*spt-1))/(*spt));
    *pt=trks+*off;
    return 1;
}



/*
 *  return next logical track/sector
 *  replace by a call to next() if not required
 *  cp/m version sets sector number according to translate table
 */

lnext(t,s)
    int *t,*s;
{   int l;
    if (xlt) {
        l=0;
        while (xlt[l++]!=*s);
        if (l>=sectors) {
            nextt(t);
            l=0;
        }
        *s=xlt[l];
    } else next(t,s);
}

/*
 *  reset the currently selected drive
 *  used prior to any other operation on a drive
 *  no error code is returned
 */

rstdrv()
{   bios(8,0);
}

/*
 *  read sector, false if error
 */

read(trk,sec,adr)
    int trk,sec;
    char *adr;
{   bios(10,trk);
    bios(11,sec);
    bios(12,adr);
    if (bios(13,1)) return 0;
    return 1;
}

/*
 *  write sector, false if error
 */


write(trk,sec,adr)
    int trk,sec;
    char *adr;
{   bios(10,trk);
    bios(11,sec);
    bios(12,adr);
    if (bios(14,1)) return 0;
    return 1;
}

/*
 *  bios call
 *  will only work with the original small C
 *  other C compilers will usually have the argument sequence reversed
 */

bios(fun,arg)
    int fun,arg;
{   char *ofs;
    ofs=(fun-1)*3;
#asm
    pop     d           ;ofs
    pop     h           ;ret
    pop     b           ;arg
    push    b
    push    h
    push    d
    lhld    1           ;get pointer to bios
    dad     d           ;add offset
    lxi     d,retn1
    push    d
    pchl                ;go
retn1:
    xchg
    mov     l,a
    mvi     h,0
    pop     b           ;ofs
    push    b
    mov     a,c         *
    cpi     (9-1)*3     ;select disk function?
    jnz     retn2
    xchg                ;yes
retn2:
#endasm
}

/*
 *  some small C library functions need redefinition
 */

#asm
ccgo:
    ret
;
qzexit:
    jmp     0
```

End Listing

Make the Most of Your
Check Digits

Peter Zilahy Ingerman

Peter Zilahy Ingerman, Systems Consultant, 40 Needlepoint Lane, Willingboro, NJ 08046-1997.

There was a question about check digits in Dr. Dobb's Clinic. I wrote an answer to it, and was asked to produce a small paper. Here it is.

All of the programming examples in the paper are written in CBASIC.

Definition

A check digit is a digit used to provide some kind of check on the number with which it is associated; usually, the check is to assess any "damage" that may have occurred to a number between the time the number is created and the time of its being revalidated.

Depending on the way the check digit is designed, it can simply identify that there has, probably, been an error, or several errors; it may be able to indicate what the error or errors probably were, and what the correction probably is.

Note that word "probably"! It is not possible to guarantee that a check digit will always do the job for which it was intended – it is merely possible to assign a reasonably high probability to its being able to do that job.

The Basic Underlying Concept

Check digits can be used in any number system; let's start with binary. Consider the binary representations of the numbers from 0 to 3: 00, 01, 10, and 11. Arrange these in a table:

	01	10	11
00	1	1	2
01	-	2	1
10	-	-	1

where the entries in the table are the number of bits by which the two binary representations differ. (Note that all possible combinations are in the table – it's arranged the way it is to conserve space!)

Now, consider those same four binary representations – but this time, with a parity bit added to make the number of 1's odd:

	010	100	111
001	2	2	2
010	-	2	2
100	-	-	2

Now the minimum number of bits that change between any two binary representations is 2. That means that if any single bit changes, an illegal representation will be produced. There is no information as to what bit is in error, but the fact that some single bit is in error will be known.

When we add this bit, we are taking a subset of all possible combinations of three bits, and electing to use only half of these possible combinations, in such a way that "between" every permitted combination of three bits there is a forbidden combination of three bits. However, since there is only one forbidden combination "between" every permitted combination, there is no way of telling, when an error occurs, which permitted combination was the original.

If two bits change, there will be no indication of an error. Hence, a single parity bit is a single-error-detecting check digit.

More formally, the minimum number of bits that change between any two binary representations is called the distance for that representation method. If the distance for a representation is n, the representation will be able to detect n–1 errors (where an "error" is defined as a bit-change).

Suppose we design a representation so that, between every permitted combination, there are two forbidden combinations. Now the minimum distance is three, so that this is a double-error-detecting representation. However, we might have enough confidence in our source of binary representations so that we feel confident that only one bit will have changed (that is, that there will have been at most a single error).

Then, in this representation, any forbidden combination of bits has a "nearest" permitted combination of bits, and we may choose to presume that the nearest permitted combination of bits was the original representation. If we do this, we can describe the representation as being a single-error-correcting representation.

There is a trade-off between error detection and error correction. An n-error detecting representation is an n-over-2-error-correcting representation; if we believe that at most n-over-2 errors will occur, we can correct them with such a code, but we will still have a problem if, in fact, more than n-over-2 errors occur (although we still could detect n errors). However, you can't have both. If you are detecting n errors, you must have given up correcting n-over-2 errors, and vice versa.

The first decision, then, that has to be made in designing a check digit is whether the intent is to detect errors or to attempt to correct errors. If a check digit is error-detecting, how many errors (in the sense of flipped bits) is it desired that it detect; if a check digit is error-correcting, a similar question must be answered. Check digits are usually intended as error-detection devices, since it is usually safer to refer an error to a human being for correction than to take the chance that a sufficiently small number of errors have occurred so that automatic correction is reasonable.

The Modulo 9 Check Digit

("Modulo 9" means "the remainder upon division by 9".)

Now let's turn to decimal numbers. Check digits are associated with decimal numbers to protect against undetected errors in transmission from one point to another; human beings may or may not be involved. Go to Figure 1. Note that the scan proceeds in that Figure from right to left. This makes no difference here, but will in some other cases, so it is a good idea to get used to doing it that way.

Now – how does it work? Consider a decimal number "abc" as a polynomial:

$$a*10\uparrow 2 + b*10 + c$$

Rewrite 10 in a funny way:

$$a*(9+1)\uparrow 2 + b*(9+1) + c$$

and expand this to get

$$a*9\uparrow 2 + 2*a*9 + a + b*9 + b + c$$

Hence, the remainder upon dividing the decimal number "abc" by 9 is

$$a + b + c$$

since all of the other terms in the expansion have 9 as a factor (and hence are divisible by 9).

But what is that funny line

```
IF Check.digit%>= 9 THEN
  Check.digit%=Check.digit%-9
```

Suppose, in the process of scanning N$ to calculate the check digit, we get a value in Check.digit% that is greater than 9. This means that a + b + c is itself a number greater than 9, but the remainder on division by 9 cannot be greater than 8, so the extra 9's in the remainder must be removed.

Finally, it is obvious (hopefully) why a Modulo 9 check digit is also called a sum-of-the-digits check digit.

The Modulo 11 Check Digit

Refer to Figure 2. If we write 10 as (11-1), the proof is essentially the same as the proof given above for the Modulo 9 check digit, and is left as an exercise. The Modulo 11 check digit is also called the alternating sum-of-digits check digit.

The Modulo 7 Check Digit

Airline tickets use a Modulo 7 check digit for every number that appears on the ticket. If you examine a ticket purchased from a ticket agent, you will find a check ,digit Modulo 7 suffixed to at least the ticket number, the travel agent number, and the airline number.

Years ago, in a telephone conversation, when the world was less suspicious and more willing to answer questions, an airline representative suggested that a Modulo 7 check digit was used because it was easy to calculate, and at the same time not nearly as obvious as Modulo 9 or Modulo 11.

What Kind of Errors Does a Modulus Catch?

A Modulus check digit is, by definition, the remainder after dividing the number by the modulus. Hence, a Modulus check digit will not catch an error such that the absolute value of the difference between the correct number and the number in error has a remainder of 0 when divided by the modulus.

Another way of looking at this is to observe that since the check digit is the remainder after division by the modulus, adding any integer multiple of the modulus to the original number will make no change in the check digit. Hence a Modulo 9 check digit will not detect the interchange of any two digits in the original number, since the absolute value of the difference of two numbers differing only by having two digits interchanged is necessarily divisible by 9 (the proof is left as an exercise). Also, it follows from the above discussion that if the modulus being used has a prime factor in common with the base of the number system, there is less information (possibly none) in the check digit than there would be otherwise.

As an illustration, a Modulo 2 or 5 check digit for a number base 10 depends only on the least significant digit of the number, a Modulo 8 check digit for a number base 10 depends only on the three least significant digits of the number, and, in general, if b is the base of the number system, and k is the smallest integer such that b ↑ k is evenly divisible by r, then a check digit modulo r for a number base b will depend only on the least significant k digits of the number.

By the same argument, a Modulo 6 check digit has only a little more information in it than a Modulo 3 check digit, since the part of the result that depends on the factor of 2 in the modulus depends only on the least significant digit of the number.

A Parenthetical Note

If N$ is the number whose check digit Modulo X is to be calculated, there are two obvious ways of calculating the check digit for a number N$:

```
T=VAL(N$)/X
Check.digit%=X*(T-INT(T))
```

which may have a problem with rounding errors under some odd circumstances, but is, otherwise, very simple indeed, and

```
T=VAL(N$)
WHILE T>X
      T=T-X
WEND
Check.digit%=T
```

which does not have any rounding error problems unless T is very large, but does require a loop whose time increases with

Figure 1

```
If N$ is the number whose check digit is to be calculated, then:


        Check.digit%=0
        IF LEN(N$)=0 THEN N$="0"
                REM Take care of an attempt to check the null string
        FOR I%=LEN(N$) TO 1 STEP -1
                Check.digit%=Check.digit%+VAL(MID$(N$,I%,1))
                IF Check.digit%>=9 THEN Check.digit%=Check.digit%-9
        NEXT I%


will leave the check digit Modulo 9 in Check.digit%.
```

Figure 2

```
If N$ is the number whose check digit is to be calculated, then:


        Check.digit%=0
        IF LEN(N$)=0 THEN N$="00"
                REM Take care of an attempt to check the null string
        IF LEN(N$) AND 1 THEN N$="0"+N$
                REM Make the length of N$ even
        FOR I%=LEN(N$) TO 1 STEP -2
                Check.digit%=Check.digit%+VAL(MID$(N$,I%,1))- \
                                VAL(MID$(N$,I%-1,1)
                IF Check.digit%>=11 THEN Check.digit%=Check.digit%-11
                IF Check.digit%<0 THEN Check.digit%=Check.digit%+11
        NEXT I%


will leave the check digit Modulo 11 in Check.digit%.
```

the number whose check digit is being calculated.

The programs given in the previous sections for calculating the check digits do not depend on the number of significant digits that can be represented in a number, since the numbers whose check digits are being calculated are treated as strings, and the only limitation is the maximum size of a string (which is 255 bytes in CBASIC; obviously much greater than the precision of a real number in CBASIC!). Hence, in the most general case, it is advisable to treat the number whose check digit is being found as a string, but if additional information is available (e.g., all of the numbers are fewer than 7 digits), it may be reasonable to use a less general algorithm.

Weighted Check Digits

Another way of computing check digits is to "weight" the digits in the original representation. An example is shown in Figure 3, where each digit is multiplied by its weight and then the weights are summed and a check digit computed from the sum of the weights. If N$ is the number whose check digit is to be calculated, W$ is one period of the weighting vector (for the example above, W$ would be "13"), and X is the modulus to which the check digit is to be computed, then the example in Figure 4 will leave the Modulo X weighted-sum check digit in Check.digit%.

Various weights have been used. Among the more common are:

. 12
. 21
. 13
. 31
. 135
. 137

Some Comments on the Weighted Check-Sum

An analysis of the weighted check-sum is, frankly, more involved than makes sense to try to include in this paper. However, it is definitely worth making some observations about ways to improve the checking capability of the technique.

First, the longer the period, the better the checking. In this sense, the three-character weights "135" and "137" are better than any of the two-character weights given above. Digits in the weight that are not factors of the base are better than digits that are. Hence the weights "12", "21", and "135" are not as good as the weights "13", "31", and "137", assuming that you are working with decimal numbers. There is less checking if adjacent weight digits are the same than if they are different. Hence, "1337" is not as good as "1373".

Figure 3

```
1  6  7  -  3  2  3  -  8  1  2      orisinal number
3  1  3  -  1  3  1  -  3  1  3      weisht
3  6 21  -  3  6  3  - 24  1  6      weishted disits
                                 76   sum of weishted disits
                                  4   76 Modulo 9 (e.s.)
```

Multiple Check Digits

Well, why not? But if you can afford to have multiple check digits, where should you put them, and how should they be calculated?

One scheme is to put them, one at a time, at the end of the number, with the calculation for each check digit including all of the check digits that have already been added. If each check digit is calculated with respect to a different modulus, this technique has about the same effect as calculating a single "check digit" to a modulus that is the product of all of the individual moduli used to calculate the individual check digits.

On the other hand, if it is possible to distribute the check digits through the original number, it may be possible to identify the digit in the original number that is in error. As an example, consider the number

1 7 2 3 5

Suppose that we insert a check digit Modulo 9 after each digit of the original number, based on all of the digits (including check digits) to the left of the check digit we are inserting:

1 7 2 3 5
117 2 3 5
11702 3 5
1170223 5
117022375
1170223751

and suppose that the 2 in the original number is changed to a 3. This would produce the number 1170333055. If this is compared with the previous number,

1170223751
1170333055
xx x x

it is clear that the left-most "x" marks the spot where there is a difference, and hence where the error occurred.

A Closing, but Parenthetical, Comment

In everything that has been discussed above, the assumption was made that the thing being examined was the representation of a number. In the programming examples, the number was represented as a string.

An ASCII character has associated with it a numeric value between 0 and 127. Hence, a string can be thought of as a representation of a number base 128. Exactly the same discussion applies to the creation of check characters for strings as applies to the creation of check digits for numbers. ▶▶J

Figure 4

```
T=0
IF LEN(N$)=0 THEN N$="0"
        REM Take care of an attempt to check the null strins
T$=""
WHILE LEN(T$)<LEN(N$)
        T$=W$+T$
WEND
        REM This makes the weishtins vector lons enoush
FOR I%=LEN(N$) TO 1 STEP -1
        T=T+VAL(MID$(N$,I%,1))*VAL(MID$(T$,I%,1))
NEXT I%
T=VAL(N$)/X
Check.disit%=X*(T-INT(T))
        REM This is a simplified computation, see above
```

A Poor Person's Spelling Checker

by Alan Bomberger

Alan Bomberger, 3721 Starr King Circle, Palo Alto, CA 94306.

This is a very simple, stand-alone spelling checker that will proofread any CP/M text file including those produced by most word processors. This program was intended to be the low-cost answer to early expensive spelling checkers, but recent announcements of a plethora of modestly priced spelling checkers have made marketing of the Speller impractical. I have decided to publish this version for non-commercial use by those interested in investigating the problems of spelling verification.

The Speller is coded in Z80 assembly language rather than 8080 assembly language, because the Z80 instruction set is much closer to that of the large IBM systems I am used to. The techniques used in this program are adaptations of ones used with large IBM systems and, because of the limitations of the Z80 instruction set, may seem awkward to the Z80 programmer.

The Speller first reads a text file, isolates words, and builds a list in memory of distinct words. Words are isolated using two tables, one of delimiters and one of letters. As each word is isolated, the word list is searched and if the word is not present it is added. Since only one list entry is built for each distinct word, a great deal of text can be processed in a single pass. If there are more distinct words than will fit in the available memory, multiple passes are made until all the text is processed. Words longer than 30 characters are split into 30 character pieces and a console message is printed identifying the long word.

After the words have been isolated from the text, lexicons are scanned and each word in each lexicon is compared with the words isolated from the text. Each correctly spelled word is flagged and after all lexicons have been scanned, the unflagged words are listed at the console in the order they appeared in the text. Lexicons are maintained with regular editors and need not be in alphabetical order.

This is a slow process and may not be practical for documents with more than 3000 distinct words. Such documents, however, are very rare because the number of distinct words used by any particular writing style is usually less than 1000.

Use

The Speller is executed with the command

[DSK:]SPELL [DSK:]FILE.TYP

After the text has been completely scanned or when the available memory is full, the user is prompted to enter the names of lexicons to scan. After all the desired lexicons have been scanned, words not located in the lexicons are listed at the console in the order in which they were isolated in the text. If there was insufficient memory to process the entire file, additional isolation and lexicon scan passes are made. There is no interactive interface with the Speller, and after all the possibly misspelled words are listed on the console, it is the responsibility of the user to make corrections in the text using whatever text editor is preferred.

All lexicon files are of the type LEX. If a <CR> is entered, then the list of possibly misspelled words will be printed on the console.

Lexicons

Lexicons are lists of words that are assumed to be correctly spelled. Each word is separated by <CR> or <CR-LF> with blanks ignored. A lexicon can be built by entering each word on a separate line of a text file. Lexicons that contain words with characters other than the 26 letters of the alphabet must be carefully coordinated with the word isolation tables described later.

In this implementation, lexicons may be compacted using a technique based on the frequency of occurrence of the letters in the lexicon. The most common 15 letters (including <CR>) are coded as 4-bit numbers from 0 to 0E and the rest of the letters (including CNTL Z) are coded as 8-bit numbers from F0 to FC. The frequency of occurrence of each letter was determined by counting the letters in the lexicon I use (33,000 words of a general vocabulary) and the tables that describe the mapping of letters to numbers are labeled T1 and T2 in the code. These tables are easily changed to reflect the relative frequency of letters for particular lexicons provided that a corresponding compression program is written to compact the lexicon. The first byte of a compacted lexicon contains an FF so that compacted and non-compacted lexicons can be recognized automatically by the Speller. This compaction technique cuts the size of a 33,000 word lexicon by one half without introducing much overhead. Other schemes can be devised with much higher compaction but may rely on the order of the words.

Since no analysis of words is performed by the Speller, all forms of words that might be used must appear in the lexicon. With this simple scheme, a lexicon would have to be extremely large in order to achieve complete spelling verification. However, including the most common forms of words makes it possible for modest sized lexicons to significantly reduce the number of words that must be manually checked.

The choice of a sequential lexicon is a tradeoff between coding complexity and speed. An indexed lexicon would allow lookup of each word as it is isolated from the text but would mean vastly more complex code and more difficult-to-maintain lexicons.

Word Isolation

Two tables in the Speller control the isolation of words. A character of text is classified as a delimiter if it is in the DELIMIT table or as a part of a word if it is in the ALPHA table. If the character is not in either table, it is ignored. When a delimiter byte is located the resulting word is added to the word list.

This two-table arrangement allows considerable flexibility in determining the nature of words and special word processor control characters are easily eliminated.

The word list is built in the available memory so its size is dependent on the hardware configuration. There are 4 bytes of control information associated with each word. A flag byte indicates the status of the word (spelled correctly, printed, last word in the list) and a length byte gives the length of the word. A two-byte chain pointer is maintained to the next in the list. When the word list is built the chain pointer is to the next word in the list and is redundant. However, before the lexicon is scanned, 27 separate

chains are built in the word list. Each word is put on the chain associated with the first letter of the word (the 27th chain is for all non-alphabetic characters).

The word list can be printed in the order it was built using the length information associated with each word, and can be scanned as 27 separate chains, with the chosen chain being the one associated with the first letter of a particular lexicon word. This technique of partitioning the word list cuts the scan time considerably. For small text files the time required to proofread is limited by the time required to scan the lexicon. For larger text files, the building of the word list becomes the limiting factor. As long as the word list does not become too large there is no serious problem, but when the word list approaches 1000 words (a very "full" vocabulary) the search time to determine if the word is already in the list can become rather lengthy.

Timings

Since the lexicons are scanned sequentially from beginning to end, a long time is required to check the spelling of a small text file. However, the time increases slowly for larger text files. The times given in Table 1 are for a 4 megahertz Z80 NorthStar computer with floppy disks. Only a single lexicon (33,000 words, 162,000 bytes compacted) was scanned for these timings.

Since at the time the word list is scanned it is divided into chains for each letter of the alphabet, the time required to scan the word list is dependent on the number of chains. If most of the words begin with the same letter, the scan time will be longer than if the words are evenly distributed on 26 chains.

When this Speller was directed to scan its own lexicon as a text file, the weakness of its simplicity became obvious. Each of the 33,000 words is, of course, distinct and only 3500 of them fit in the memory available on my 64K North Star computer. Each pass required more than 40 minutes to complete and 9 passes were required. Such an extreme case should not be the criterion for design of a particular program.

Figure 1

File size in bytes	Number of words	Time in Minutes
7	1	2
3600	250	2.6
9200	456	2.6
8200	448	2.7
24000	690	4

Summary

This Speller is both functional and provides the basis for much more complex programs. With very few exceptions no information is carried around in registers so the addition of new functions is easy. The addition of word-structure analysis routines would increase the effectiveness of small lexicons by making possible the recognition of root words with standard suffixes and prefixes. Only the root words would be in the lexicon. The addition of an "indexed" lexicon would make possible the correction of words as the text is being read. Spelling correction is possible if a "nearness" function is substituted for the simple exact match.

It is clear there is no mystery associated with the problem of proofreading a text file. There are as many ways to organize the program and the lexicon files as there are programmers and this program illustrates only one of these organizations. The principles behind this program were those of simplicity, ease of programming, and flexibility.

For those interested, I can supply machine-readable source on a North Star Double Density disk or on an 8-inch single density (IBM standard) disk for $10. A 33,000-word compacted lexicon is also availabe in the same formats for $30.

DDJ

(Listing begins on page 46)

Spelling Checker (Text begins on page 42)

```
.z80
;
;       Poor Person's Spelling Checker
;
;       Released for NON COMMERCIAL USE ONLY
;          (c)  1981  Alan Bomberger
;
;       The input file is checked using the lexicon
;       files and misspelled words are printed in the
;       order they appear in the text.
;
;       Each lexicon is scanned for each word list
;       and may be in any order.
;
;       The word list will fill all available memory so
;       only very large documents will require more than
;       one pass of the lexicons.
;
boot    equ     0
bdos    equ     5
pstring equ     9
rstring equ     10
conout  equ     2
openf   equ     15
closef  equ     16
readf   equ     20
setdma  equ     26
cr      equ     0dh
lf      equ     0ah
bell    equ     07h
fcb     equ     05ch
fcb2    equ     06ch
buff1   equ     080h
;


spelit:
        ld      sp,stack        ; a new stack pointer
        ld      de,copyr
        call    print
        call    openin          ; open input files
        call    zchn            ; zap chains
buildl:
        call    gword           ; get the next word of text
        jr      c,endin         ; no more left, check spelling
        call    search          ; see if in word list
        jr      nc,buildl       ; yes it is
        call    addw            ; no, so add it in
;
;       "work"  contains the address of the last word
;       put into the word list.  See if this word is passed the
;       threshold of memory
;
        call    compare
        jr      nc,buildl       ; no so continue
        ld      hl,numwdc+2     ; mark as incomplete
        ld      (hl),"$"        ;
        call    spell           ; check the current list
        call    ptable          ; print the misspelled words
        ld      ix,comwdl       ; the last of the common words
        set     wflgsl,(ix+wflgs)       ; mark this as the last in list
        ld      c,6
        ld      b,0
        ld      de,numwd
        ld      hl,zcount       ; zero counter
        ldir
        call    zchn            ; zap chains
        jr      buildl          ; and get next word
endin:
        call    spell           ; check the spelling of words in list
        call    ptable          ; print the ones not in lexicon
        call    closein         ; close up input file
        jp      boot            ; return
;
;       compare
;
;       compare the value in work with "endmem"
;
compare:
        ld      hl,work         ; address of last word
        ld      a,(endmem)      ; end of memory
        sub     (hl)
        ld      a,(endmem+1)
        inc     hl
        sbc     a,(hl)          ; double precision subtract
        ret
;
;       openin
;
;       open input file and locate end of memory
;
openin:
        ld      de,fcb          ; input file
        call    open
        ld      de,ninput       ; in case not there
        inc     a
        jr      z,failed        ; no file
        ld      a,128
        ld      (ibp),a         ; set so first call gets disk record
;
;       find end of memory
;
        ld      hl,(6)          ; address of bdos
        ld      bc,64           ; a margin
        or      a               ; clear carry
        sbc     hl,bc           ; subtract margin
        ld      (endmem),hl
        ret
failed:
        call    print
        jp      boot            ; quit now
;
;       gword   get next word in text into cword
;
;       carry flag on means end of input
;
gword:
        ld      a,128
        ld      (cflags),a      ; set this word as last
        ld      de,0            ; length of word
gwordl:
        call    gbyte           ; get next byte of text
        jr      c,gworde        ; end of input
        ld      bc,(deliml)     ; length of delimiter table
        ld      hl,delimt       ; the table
        cpir                    ; is it a delimiter
        jr      z,delim         ; yes
        ld      bc,(alphal)
        ld      hl,alpha        ; is it alphabetic
        cpir
        jr      nz,gwordl       ; no skip it
        cp      "a"             ; is it lower case
        jr      c,gwordu        ; no
```

```
        cp      "{"             ; lower
        jr      nc,gwordu
        and     05fh            ; make all upper case
gwordu:
        ld      hl,cword+4      ; place to build word
        add     hl,de
        ld      (hl),a          ; put byte in word
        inc     e               ; new length
        ld      a,e
        ld      (clen),a        ; update in word entry
        cp      30              ; how long is word
        jr      z,gwordt        ; too long a word
        jr      gword1          ; loop
delim:
        ld      a,e             ; current length
        cp      0
        jr      z,gword1        ; skip leading delimiters
        or      a               ; zero carry flag
gworde:
        ret
gwordt:
        ld      de,lngwd1       ; first part of text
        call    print
        ld      de,cword+4
        call    print
        ld      de,lnglx2       ; second part
        call    print
        or      a
        jr      gworde
;
;       getbyte
;
;       get next byte of text
;       carry flag on for end of file
;
gbyte:
        push    de
        ld      a,(ibp)
        cp      128             ; do we need another buffer full
        jr      nz,gbyte1       ; no
        ld      de,fcb
        call    diskr           ; read a block
        cp      0               ; did it ok?
        scf                     ; in case not
        jr      nz,gbyter       ; end of file return
gbyte1:
        ld      e,a             ; has current byte index to fetch
        ld      d,0             ; double precision
        ld      hl,buff1
        add     hl,de
        inc     a               ; next index
        ld      (ibp),a
        ld      a,(hl)          ; get byte
        cp      01ah            ; check for end
        scf                     ; in case it is
        jr      z,gbyter        ; yes
        or      a               ; clear carry flag
gbyter:
        pop     de
        ret
;
;       search
;
;       search word list for match with cword
;
;       on return ix will point to matched entry or last in list




;       carry on if no match
;
;       searc1 is the entry when searching on a chain
;
;
search:
        ld      ix,words        ; start of list
searc1:                         ; entry if starting with chain
sloop:
        ld      a,(clen)        ; length of current word
        cp      (ix+wlen)       ; must be same as list entry
        jr      nz,nextw        ; try next entry
        call    clc             ; compare
        jr      z,match         ; it is a match
nextw:
        bit     wflgs1,(ix+wflgs)     ; is this last entry
        jr      nz,nmatch       ; yes then no match
        ld      a,(ix+wchn)     ; get chain pointer
        ld      (work),a
        ld      a,(ix+wchn1)    ; both parts
        ld      (work+1),a
        cp      0               ; this is high order (never zero unless end)
        jr      z,nmatch        ; end of chain
        ld      ix,(work)
        jr      sloop
match:
        or      a               ; clear carry
        jr      sret
nmatch:
        scf                     ; set carry
sret:
        ret
;
;       clc
;
;       compare logical character
;       cword with list entry pointed to by ix
;       a contains length
;
clc:
        push    ix
        ld      c,a             ; length for down count
        ld      hl,cword+4      ; compare here
clc1:
        ld      a,(ix+word)     ; first character
        cp      (hl)            ; is it
        jr      nz,clce         ; no stop
        inc     hl
        inc     ix
        dec     c
        jr      nz,clc1         ; not end so continue
clce:
        pop     ix
        ret
;
;       addw
;
;       add word to list
;       word is in cword and ix points to last entry
;
addw:
        ld      (work),ix               ; save
        ld      iy,(work)               ; old position
        ld      a,0
        ld      (cchn),a
        ld      (cchn1),a               ; zero chain pointer


        res     wflgs1,(ix+wflgs)       ; clear this is last entry flag
        ld      b,0
        ld      a,(ix+wlen)             ; get length of last word
```

(Continued on next page)

Spelling Checker (Listing continued, text begins on page 42)

```
        add     a,4
        ld      c,a                     ; include chain and stuff
        add     ix,bc                   ; skip over last entry
        ld      (work),ix
        ld      a,(work)                ; get low byte
        ld      (iy+wchn),a             ; to chain
        ld      a,(work+1)
        ld      (iy+wchn1),a            ; to chain
        ld      a,(clen)
        add     a,4
        ld      c,a
        ld      hl,cword        ; source
        ld      (work),ix
        ld      de,(work)       ; cant get there from here
        ldir                    ; move it
        call    countw          ; bump count
        ret
;
;       spell
;
;       check each lexicon word with list entries
;       mark correct (found) words in list
;
spell:
        ld      de,numwd
        call    print           ; inform of number of words
        call    setchn          ; set up chains
        ld      de,buff2        ; switch buffers
        call    setbuf
nextlex:
        call    getlex          ; get a lexicon file
        jr      c,spellr        ; none so return
        ld      de,lfcb         ; get lexicon file
        call    open
        ld      de,nolex        ; in case not there
        inc     a
        jp      nz,gotlex       ; it is a valid lexicon
        call    print           ; it is not a valid lexicon
        jr      nextlex         ; try again
gotlex:
        ld      de,lfcb         ; lexicon fcb
        call    diskr           ; read first record
        cp      0               ; did it
        jr      nz,endl         ; quick exit
        ld      de,checkm       ; tell customer
        call    print           ; that we begin
        ld      a,0
        ld      (ibpl),a
        ld      (comp),a        ; say not compacted
        ld      a,(buff2)       ; first of compacted
        cp      0ffh
        jr      nz,spelll
        ld      a,1
        ld      (comp),a        ; set compacted
        ld      (ibpl),a        ; skip  ff
spelll:
        call    lword           ; get a word in cword
        jr      c,endl          ; end of lexicon
        ld      ix,cword
        call    getchn          ; get correct chain for this word
        ld      e,(hl)          ; low order byte
        inc     hl
        ld      d,(hl)          ; high order byte

        ld      (work),de       ; get first word in list
        ld      ix,(work)       ; place to start
        ld      a,(work+1)
        cp      0
        jr      z,spelll        ; if zero no words this letter
        call    searc1          ; look for word in chain
        jr      c,spelll        ; did not find it
        set     wflgsc,(ix+wflgs)       ; mark spelled correctly
        jr      spelll          ; and loop
endl:
        ld      de,lfcb         ; close
        call    close
        jr      nextlex         ; get another lexicon
spellr:
        ld      de,buff1        ; reset dma
        call    setbuf          ; in case more input
        ret
;
;       getlex
;
;       get a lexicon file from the customer
;       if none requested (null input) return with carry flag on
;
getlex:
        ld      de,asklex
        call    print           ; type prompt
        call    answer          ; get answer
        jr      c,getlxr        ; return . no lexicon
        call    bldfcb          ; build a new fcb
        or      a               ; clear carry flag
getlxr:
        ret
;
;       answer
;
;       get answer to question in buff2
;
answer:
        ld      de,buff2
        ld      a,80
        ld      (buff2),a
        call    input           ; get answer
        ld      a,(buff2+1)     ; get length of answer
        cp      0               ; see if any
        scf                     ; none
        jr      z,answrt        ; quit now
        or      a               ; clear carry flag


answrt:
        ret
;
;       bldfcb
;
;       build an fcb from information in buff2
;       assume file type of .LEX

;
bldfcb:
        ld      hl,deffcb       ; the default fcb
        ld      de,lfcb         ; goes here
        ld      bc,16           ; move this much
        ldir                    ; move it
        xor     a               ; get a zero
        ld      (lfcbcr),a      ; zero this as well
        ld      hl,buff2+2
        ld      a,(buff2+1)     ; get number of bytes in name
        ld      c,a             ; b is zero from block move
```

```
bllloop:
        ld      a,(hl)          ; get a byte
        cp      " "             ; is it a blank
        jr      nz,noblk        ; no
        inc     hl
        dec     c
        jr      nz,bllloop      ; skip leading blanks
        jr      bldret          ; return with bad fcb
noblk:
        inc     hl              ; skip disk name if present
        ld      a,(hl)          ; get suspected ":"
        dec     hl              ; back to first character
        cp      ":"             ; is it a disk name
        jr      nz,nodsk        ; no just a name
        ld      a,(hl)          ; get disk name
        and     0fh             ; to cp/m standards
        ld      (lfcbdn),a      ; to fcb
        inc     hl
        inc     hl              ; skip name and ":"
        dec     c
        jr      z,bldret        ; quit with bad fcb
        dec     c
        jr      z,bldret        ; quit with bad fcb
nodsk:
        ld      de,lfcbfn       ; place for name
        ld      a,8             ; max length at this point
        cp      c               ; are we ok
        jr      c,bldret        ; no so leave blank
filelp:
        ld      a,(hl)
        cp      "."             ; this is end (we ignore)
        jr      z,bldret
        cp      " "             ; also end
        jr      z,bldret        ; and this
        cp      "a"             ; lower case alpha
        jr      c,filel1        ; no
        and     05fh            ; make upper
filel1:
        ld      (de),a          ; put in fcb
        inc     de
        inc     hl
        dec     c
        jr      nz,filelp       ; loop
bldret:
        ret
;
;       lword
;
;       get a lexicon word
;       carry flag on if end of lexicon
;
lword:
        ld      de,0            ; length of word
lwordl:
        call    lchar           ; get char from file
        jr      c,lwordr        ; if end
        cp      1f              ; skip these if present
        jr      z,lwordl
        cp      " "
        jr      z,lwordl        ; skip blanks in lexicon
        cp      cr              ; end of word
        jr      z,lworde        ; done
        cp      01ah            ; end
        jr      z,lwordf        ; set carry and return
        cp      'a'             ; lower case?
        jr      c,lwordu        ; no, upper
        cp      "{"


        jr      nc,lwordu
        and     05fh            ; make sure upper case
lwordu:
        ld      hl,cword+4      ; place to put it
        add     hl,de
        ld      (hl),a          ; build word
        inc     e               ; bump count
        ld      a,e
        ld      (clen),a
        cp      30              ; how long
        jr      z,lwordt        ; too long
        jr      lwordl          ; get more bytes
lworde:
        ld      a,e             ; check for null word
        cp      0               ; any so far
        jr      z,lwordl        ; no so continue
        or      a               ; clear carry
lwordr:
        ret
lwordf:
        scf
        jr      lwordr          ; return
lwordt:
        ld      de,lnglx1       ; first part
        call    print
        ld      de,cword+4
        call    print
        ld      de,lnglx2       ; second part
        call    print
        or      a
        jr      lwordr
;
;       lchar
;
;       get a character from lexicon (compacted or not)
;
lchar:
        ld      a,(comp)        ; is it compacted lexicon
        cp      0               ; well?
        jr      nz,lcharc       ; yes
        call    lbyte           ; no get a byte
        ret                     ; and return
;
lcharc:
        call    gnib            ; get a nibble
        jr      c,lchare        ; end already
        cp      0fh             ; is it flag
        jr      z,lchars        ; yes second set of letters
        ld      c,16            ; size of table
        ld      hl,t1           ; in table one
lchar1:
        cp      c
        jr      nc,lchare       ; too big
        ld      b,0
        ld      c,a
        add     hl,bc
        jr      lcharg          ; got it
lchare:
        ld      de,badlex
        call    print
        scf
        ret                     ; say end of lexicon
lchars:
        call    gnib
        jr      c,lchare
        ld      c,14            ; search length


        ld      hl,t2
        jr      lchar1          ; loop here
lcharg:
        ld      a,(hl)
```

(Continued on next page)

Spelling Checker (Listing continued, text begins on page 42)

```
        or      a               ; clear carry
        ret
;
;       gnib
;
;       get a nibble from compacted lexicon
;
gnib:
        ld      a,(lrnib)
        cp      1               ; left or right
        jr      z,gnibr         ; right
        ld      a,1
        ld      (lrnib),a
        call    lbyte           ; get a byte
        jr      c,gnibr         ; report carry
        ld      (byte),a
        srl     a
        srl     a
        srl     a
        srl     a               ; put left in lower
        or      a               ; clear carry
        ret
gnibr:
        ld      a,0
        ld      (lrnib),a
        ld      a,(byte)
        and     0fh
        ret
;
;       lbyte
;
;       get a byte from lexicon file
;       carry flag on for end of file
;
lbyte:
        push    de
        ld      a,(ibpl)        ; get buffer pointer
        cp      128             ; at end
        jr      nz,lbyte1       ; no
        ld      de,lfcb         ; fcb for lexicon
        call    diskr
        cp      0               ; did it work
        scf                     ; in case not
        jr      nz,lbyter       ; return with carry if end
lbyte1:
        ld      e,a             ; position in buffer
        ld      d,0
        ld      hl,buff2
        add     hl,de           ; correct byte
        inc     a               ; for next time
        ld      (ibpl),a
        ld      a,(hl)          ; get the byte
        or      a               ; clear carry flag
lbyter:
        pop     de
        ret
;
;       ptable
;
;       print misspelled words from list
;
ptable:



        ld      b,0
        ld      ix,words        ; start
ptloop:
        bit     wflgsc,(ix+wflgs)     ; is this one correct
        jr      nz,pnext        ; yes don't print it
        call    pword           ; print the word
pnext:
        bit     wflgsl,(ix+wflgs)
        jr      nz,ptabr
        ld      a,(ix+wlen)     ; get length this entry
        add     a,4
        ld      c,a
        add     ix,bc
        jr      ptloop          ; try again
ptabr:
        ret
;
;       pword
;
;       print word pointed to by ix
;
pword:
        push    ix
        ld      b,(ix+wlen)
pwloop:
        ld      e,(ix+word)     ; a character
        push    bc
        call    pchar
        pop     bc
        dec     b
        jr      z,crlf
        inc     ix              ; next character
        jr      pwloop
crlf:
        ld      e,cr
        call    pchar
        ld      e,lf
        call    pchar
        pop     ix
        ret
;
;       count words

;
countw:
        ld      hl,numwdc       ; get lowest byte
        ld      a,":"           ; a test for too large
count1:
        inc     (hl)
        cp      (hl)            ; see if too big
        ret     nz              ; no
        ld      (hl),"0"        ; yes set to 0
        dec     hl
        jr      count1          ; backup and try again


;
;       zchn
;
;       zero chain headers
;
zchn:
        ld      a,0             ; get a zero
        ld      c,054d          ; number
        ld      hl,alphc        ; place
zchn1:
        ld      (hl),0
        inc     hl
        dec     c
```

```
        jr      nz,zchnl
        ret
;
;       getchn
;
;       get address of chain head of word pointed to by ix
;
getchn:
        ld      a,(ix+word)     ; first char
        ld      b,0
        ld      hl,alphc        ; first chain head
        cp      'A'             ; first
        jr      c,chnoth        ; lower use other
        cp      '['
        jr      nc,chnoth       ; greater use other
getcha:
        and     01fh            ; mask
        dec     a
        sla     a               ; double it
        ld      c,a             ; displacement
        add     hl,bc
        ret
chnoth:
        ld      a,'['
        jr      getcha          ; use last chain
;
;       setchn
;
;       scans word list and rechains it by letter



;
setchn:
        ld      ix,words        ; place to start
setch0:
        call    getchn          ; get the correct header
        ld      a,0             ; get a zero
setch1:
        inc     hl              ; to high order byte
        cp      (hl)
        jr      nz,nxtchn       ; not this one
        ld      (work),ix       ; goes here
        ld      de,(work)       ; get it
        ld      (hl),d
        dec     hl
        ld      (hl),e
        ld      (ix+wchn),a     ; zero forward
        ld      (ix+wchn1),a
        jr      setchw          ; next word



nxtchn:
        ld      d,(hl)
        dec     hl
        ld      e,(hl)
        ex      de,hl
        inc     hl
        inc     hl              ; to chain portion of word
        jr      setch1
setchw:
        bit     wflgs1,(ix+wflgs)
        jr      nz,setchr       ; return
        ld      a,(ix+wlen)
        add     a,4
        ld      c,a
        ld      b,0
        add     ix,bc
        jr      setch0
setchr:
        ret
```

```
;
closein:
        ld      de,fcb
        call    close
        ret
;
open:
        ld      c,openf
        jp      bdos

close:
        ld      c,closef
        jp      bdos
print:
        ld      c,pstring
        jp      bdos

diskr:
        ld      c,readf
        jp      bdos
pchar:
        ld      c,conout
        jp      bdos
setbuf:
        ld      c,setdma
        jp      bdos
input:
        ld      c,rstring
        jp      bdos
;

;
        ds      64
stack:  ds      1
endmem: ds      2
deffcb: db      0,"       LEX",0,0,0,0
lfcb:   ds      33
lfcbcr  equ     lfcb+32
lfcbex  equ     lfcb+12
lfcbs1  equ     lfcb+13
lfcbs2  equ     lfcb+14
lfcbrc  equ     lfcb+15
lfcbdn  equ     lfcb+0
lfcbfn  equ     lfcb+1
lfcbft  equ     lfcb+9
ibp:    ds      1
ibpl:   ds      1
work:   ds      2
byte:   ds      1
lrnib:  db      0
comp:   db      0
buff2:  ds      128
zcount: db      "0000  "
numwd:  db      "0000   distinct words in text",cr,lf,"$"
numwdc  equ     numwd+3
lnglx1: db      "lexicon word '$"
lnglx2: db      "' longer than 29 characters",cr,lf,"$"
lngwd1: db      "text word '$"
badlex: db      "error in compacted lexicon",cr,lf,"$"
ninput: db      "input file not specified or non-existant",cr,lf,"$"
nolex:  db      cr,lf,"lexicon file not specified or non-existant",cr,lf,"$"
checkm: db      cr,lf,"begin spelling check pass",cr,lf,"$"
asklex: db      "enter lexicon file name (.LEX  assumed) or 'return'"
        db      bell,cr,lf,"$"
copyr:  db      cr,lf,"Poor Person Speller (c) 1981, Alan Bomberger"
        db      cr,lf,cr,lf,"$"
cword:  ds      34
        db      "$"
cflags  equ     cword
```

(Listing continued on page 53)

Spelling Checker

(Listing continued, text begins on page 42)

```
clen    equ     cword+1
cchn    equ     cword+2
cchn1   equ     cword+3
wflgs   equ     0
wlen    equ     1
wchn    equ     2
wchn1   equ     3
word    equ     4
wflgsl  equ     7
wflgsc  equ     6
wflgsp  equ     5
delimt: db      ".,;:()!?[] /'{}-",cr,lf,34,9,"\"
        db      0,0,0,0,0,0,0,0
deliml: db      deliml-delimt-8,0
alpha:  db      "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        db      "abcdefghijklmnopqrstuvwxyz"
        db      0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
alphal: db      alphal-alpha-20,0
t1:
        db      'EISNATR'
        db      'OLDCUGP',cr
t2:     db      'MHBYFVW'
        db      'KZXQJ',01ah
alphc:  ds      54
words:
comwdl:
        db     192,1,0,0,"A"
        end     spelit
```

(End Listing)

Son of Runic

Writing the Compiler

by Marty Franz

In my last article, "Runic: An Interactive, Extensible Compiler" (*DDJ* #65), I described a FORTH-like programming language called Runic. It featured high-level data types, a dictionary that allowed forward-referencing definitions, and Pascal-like logic structures. I deliberately kept implementation details out of that article because when I set to work I did not have a specific computer in mind – all I knew was that I wanted a language very much like Runic for my own use.

After laying out specifications for the language more or less as they appeared in my last article, I began to develop Runic in July of 1979 on my PET computer. After months of futilely trying to write the tools I needed – such as a text editor and assembler – I sold the PET and bought a CP/M-based system instead. The PET may be fine for some things, but language development isn't one of them.

CP/M, despite its expense, gave me the development environment I needed. With my choice of tools made, my next step was to write a BASIC model of the compiler. Doing so, I felt, would not only help make me more familiar with CP/M but would identify problems early on in my development effort. That it did, but before writing a model I had to settle on a design approach. Two were worth considering:

1. I could make Runic a "pure interpreter," directly executing source code. This would be slow, but simple. There's also no need for a decompiler to make listings, since the source code is always kept around.
2. I could make Runic a "compiler/interpreter," like most BASIC's, compiling source lines into an intermediate object language first and then interpreting that. This would make Runic run much faster, but the compiler would be more complicated. In addition, you need a decompiler function to get the source code back again.

It wasn't until I tried to add structured logic blocks to the pure interpreter that I realized I would have to use the other design instead – not only was repeatedly executing blocks of source code quite slow, but just scanning them was expensive, since you had to find the end of the block each time you evaluated a conditional, independent of whether the block was executed or not.

Once this decision was made, I set to work. I invested a weekend writing BASIC models of the compiler and interpreter, and ran some test cases through them. The models provided useful information, especially in the area of compiling the "if" and "do" logic blocks.

After I had written the "toy" version of the compiler, I was ready for the real one. I needed a good high-level implementation language – assembler, I felt, would be too slow to develop in. At the urging of my friends I tried C, and was hooked. It was easy to use, efficient and, best of all, the source code was quite readable.

From September, 1980, through April, 1981, I wrote Runic's compiler, mostly on weekends and at night. At the Sixth West Coast Computer Faire I brought along (and more or less gave away) demo versions. A production version is now for sale from Starside Engineering. It runs on any 48K or larger CP/M system and features over 100 words, a demo package, and a manual.

In its present form, the Runic compiler has these major components:

1. A small driver that reads source statements from either the CP/M console or an input file and calls, in turn, the compiler and executive described below. This is a fairly simple function and will not be covered further in this article.
2. The compiler that builds a string of object codes out of the statement.
3. The executive that performs the actions specified by the opcodes.
4. Kernel functions called by the executive that control the stack, condition flag, and dictionary.

Figure 1 shows how each piece relates to the other. Now we'll look at each component in more detail.

The compiler conducts the scan of a user's statement, building a set of object codes which are then executed by the executive.

The compiler first tokenizes the entire line of Runic source. A "token" is either a word in the dictionary, a numeric constant, or a quoted string. For the latter two token types, codes are generated to push the data onto the stack, and the constant data is copied directly to the object line.

Otherwise, the dictionary is searched. Every entry in the dictionary has these four fields:

1. The word's name, converted to lower case and no more than 8 characters long.
2. A flag that tells what type of word it is: number, string, kernel, or user definition.
3. An address pointer to a function called when the word is executed; or, for numbers, the value of the number itself, stored as a two-byte integer.
4. An address pointer to a function called when the word is compiled. This pointer is used for words like "if" and "do" that require extensive processing at compile time.

If the current word is found in the dictionary, then the compile-time function for it is called (if it exists) or else a one-byte index to the entry is placed in the object code. Allowing one byte for the index means that 255 entries can be present in the dictionary at one time.

If the word wasn't found, and the user was defining a new word, then a dummy entry is created. This scheme allows for forward references – at some later time a new execution address can be filled in and the original definition will remain unchanged. And this is the reason one-byte object indexes are used – they remove the object code a step from the dictionary's structure, allowing words to be re-defined later.

If an assignment operator ("^") was in front of the word, then an extra code is generated to tell the executive to pop the top of the stack into the word during execution.

Internal stacks are maintained by the compiler to keep track of the relative branch locations for "if" and "do" blocks. Relative branches are generated rather than address branches so that a definition can be moved around and copied in object form. This, and a dictionary index rather than a direct address being used as the object code, lets each dictionary entry be a separate entity that may be replaced or deleted independently of the others.

After a line has been compiled, it's marked with another index that tells its length. This is so that error tracebacks and listings can be built from just the object code and the name of the word kept in the dictionary. Since only a user defini-

Copyright © 1981 by Marty Franz, 525 W. Walnut St., Kalamazoo, MI 49007.

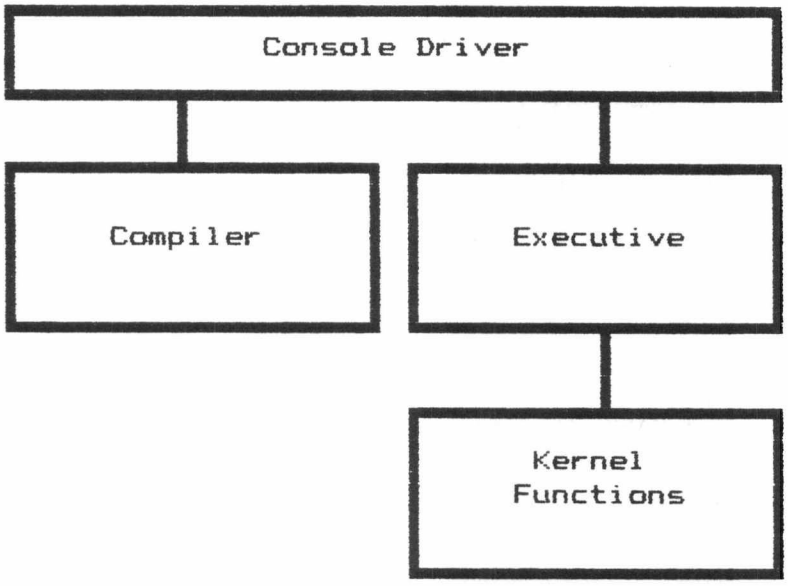


Figure 1. A Block Diagram of Runic

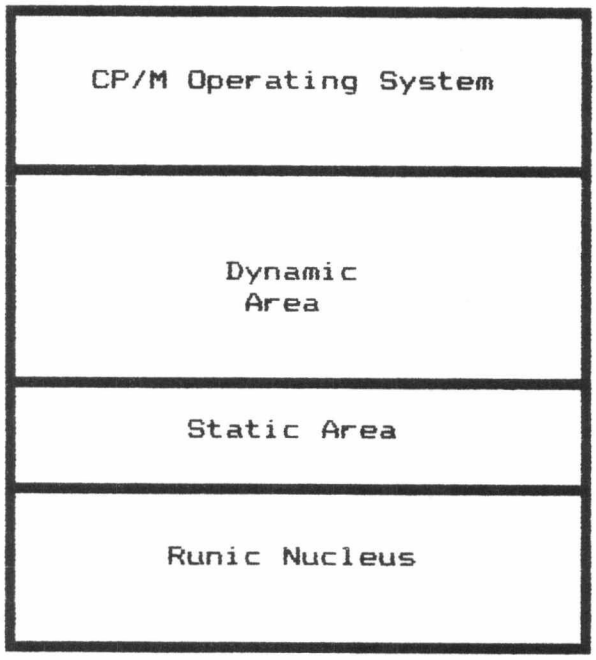


Figure 2. Runic's Memory Map

tion's object code is stored, this small code overhead is necessary to preserve a source line's integrity. Comments, if used, are stripped off during compilation and not saved with the object, as they are in BASIC. If commented source is needed, it can be kept on disk, then read and recompiled when the definition is needed.

If compilation of a direct statement was successful, the object code is executed immediately. On the other hand, if the user was defining a new word with ":", the code is saved as part of the word's dictionary entry and linked via the execution-time function pointer. In definition mode, lines are entered and compiled until the user types ";", at which point the complete entry is added to the dictionary as a new definition.

The executive takes over from the compiler, executing the object line. It uses the object codes to locate and call successive execution-time functions. Execution proceeds until a "return" word is reached (this has a special object code of zero) or until an error occurs.

The executive is started only after a direct statement is entered, not a definition. It is called recursively (an advantage to writing the compiler in C instead of Assembler) for each user definition called with that definition's execution-time address as the new program counter. It runs through object codes until a flag is set telling it to stop. This flag is set in one of three ways: (a) by executing a "return" word; (b) by Runic's error handler (such as on a stack overflow); or (c) by the user pressing ESC (the break-in key). After the executive is stopped, it decides whether a traceback is required.

If a traceback is needed (all errors that occur at execution time produce one) then the exeuctive uses the marker at the end of the current line to determine the start of the erroneous statement. It then calls the rebuilding function to display it at the user's console.

An example of an error traceback is shown in this example:

```
> "hi" 2 + say .
- Wrong type of data
- At: "hi" 2 + say .
```

Here, the user tried to add a number and a string together, hence the "– Wrong type of data" message. The rebuilding function is also used to display user definitions through the "show" word, mentioned in the last article. This capability is important in Runic, since it helps users

list and revise definitions without the need for a source editor.

Providing a break-in key is another important function of the executive. After each opcode is executed it checks CP/M's console input port to see if ESC has been pressed during the last instruction. If so, the executive is halted and a trace message displayed. This lets the user interrupt a wayward program to check the stack or its variables.

In order to perform the actions specified by the user in his statement, the executive calls kernel functions. These alter the state of the Runic machine: they push and pop stack data, set the condition flag, and control branching. Each kernel function, when called, is passed the current program counter. A kernel function's single interface requirement is that it alter this program counter in some way prior to returning to the executive so that execution can continue.

New kernel functions are added to the dictionary by the "kernel" word. This is actually a small loader called by the executive with the name of a CP/M file on top of the stack. The CP/M file contains the kernel function in relocatable format. The "kernel" word reads this new function in, relocates it, and adds its name to the dictionary. Once loaded, the new kernel is available as part of the language and can be saved with the rest of the compiler. The kernel function makes procedure calls to push and pop stack data, and must correctly maintain the program counter in order to work properly.

The stack itself is an array of three-byte entries. For numbers, the first byte contains a zero flag and the next two contain the actual integer value. For strings, the flag byte is non-zero and the next two bytes contain a pointer to the address where the string is stored. Stack limit- and data type-checking are done by the routines that directly push and pop data, with null values returned if an error is encountered. A local copy of a string is made whenever it is pushed on the stack – no attempt is made to lock or chain descriptors to cut down on copying. The stack (like the dictionary) is a fixed-size data structure allocated when the compiler is first generated.

Free memory for strings and user definitions is managed in the compiler through the use of 32-byte "cells" maintained by a bit map. The technique is similar to that used to manage disk storage: requests for free memory are rounded to a multiple of the cell size and an area is allocated out of the bit map to meet the request.

With this in mind, Runic's memory map under CP/M looks like Figure 2: above the compiler's resident code there is a fixed area containing the memory management bit maps, the dictionary, and the stack. On top of the fixed area up to the top of the TPA is a dynamic area used for user definitions, kernel functions, and strings. The size of this dynamic area is computed each time the compiler is run, based on the amount of free memory available.

What's performance like? Runic programs are about as fast as comparable ones written in BASIC. The compiler occupies 24K bytes on CP/M. Another 4K bytes is used by the standard library, a set of frequently-used user definitions and utilities.

To interface to the CP/M operating system, three I/O words, "read", "write" and "close" have been included. They provide re-direction of console I/O to CP/M files. With them, a Runic user can develop programs interactively using the compiler, then save them on disk, where they can be accessed by any CP/M-compatible text editor. A "program" is a set of user definitions loaded or saved as a group. Development using Runic is particularly easy, since the user can code top-down or bottom-up as the problem demands.

My present plan for Runic is to get the system to as many serious users as possible for their evaluation and feedback. While not as efficient as FORTH, Runic does have features that make it attractive for teaching FORTH-like programming languages to novice users, such as its flexible dictionary structure, convenient strings, break-in key, and error checking. A few sessions with Runic might be a good learning experience for a hobbyist interested in going beyond BASIC programming but perhaps not quite up to tackling one of the excellent FORTH systems currently available.

In addition to distributing the compiler, I plan to enhance it with more error recovery, kernel words, and documentation. I would appreciate suggestions on this from interested *DDJ* readers. At the very least, Runic shows that it is possible to develop serious software on today's micros. I hope readers working on similar projects will take heart from this and continue their own efforts. **DDJ**

SOFTWARE REVIEWS

PIE
Walt Bilofsky
$29.95 from
The Software Toolworks
(Reader Service No. 451)

Video Scribe
Ron Rocheleau
$39.95 from
Newline Software
(Reader Service No. 461)

Reviewed by D. C. Shoemaker

Today's personal computer owner generally has at least one of three basic purposes in mind for his or her system. The primary one may be game playing (to which few will admit, but which is a powerful lure for me), data manipulation (keeping track of information for business or home use) or text processing of one kind or another. I don't mean to exclude the systems enthusiast, often referred to as a hacker, who may actually spend more time on implementation than on the use of a piece of software or hardware, but such folk frequently have something like these goals as their final objective. Regardless of your goal, one essential piece of applications software you need is an editor. Such programs are the subject of my comments, and I will compare two full-screen type editors currently available for the Heath H89 computer (or H8 computer with H19 terminal.)

The editors are Walt Bilofsky's PIE 1.5 (The Software Toolworks #201 or Heath Softstuff SF-9000) and Ron Rocheleau's Video Scribe Ver. 3.0. Both of these are termed full-screen editors, which means they give a "page" of text on the CRT's screen with the ability to position the cursor anywhere on that page of text by using some form of cursor-positioning commands (usually generated by a special set of keys or a keypad on the terminal). Naturally, this feature may be implemented in a variety of ways, from the fairly clumsy and difficult to use (as with Heath's distribution editor that comes with HDOS, or CP/M's ED) to the smooth and logical, as in the case of PIE and Video Scribe (VS).

In addition to a controllable cursor, a good editor should have an easy way to change the page being displayed. Both PIE and VS allow one to page ahead or backwards through the text (scrolling), either a line at a time or a page of 24 lines at a time. Both allow immediate return to the head of the file, but VS also allows one to drop to the end of the file with a single command. One can get there with PIE, but by entering an additional command first.

From this point on, we're going to look at the special features one might consider "bells and whistles." One could live and work without them, but they sure make the job easier. Both PIE and VS include the capability to insert a line or a character, and to delete a line or a character. Both allow moving blocks of text, selecting a paragraph to move from the middle of a file to the end, for instance. Both allow the destination disk to be replaced, thereby offering the chance to leave the original file intact as a back-up; and both will allow deletion of the old file should there not be enough room for both to be saved on the same disk, helpful for single-disk users. In both cases, if the disk has too little space, an error message is generated. Both display the amount of space remaining in RAM (but not on disk, as Auto-Scribe does).

Both PIE and VS can be optimized for Z80 use – the former by virtue of providing two separate programs on the disk, one for the 8080 and one for the Z80; and the latter by a settable option. In the case of VS, another settable option is automatic word wrap-around (happiness is never having to hit RETURN). In PIE, you would enter CTRL-W to turn that feature on or off, but only in the Software Toolworks version 1.5b; Softstuff's version and PIE 1.5 lack that enhancement.

PIE and VS both come with excellent written documentation; PIE requires about 11 pages, and VS about 13. In addition, the Softstuff version of PIE also comes with the printed documentation on disk as a test file, so the user can replace the written copy or create an abbreviated version for quick reference.

Finally, what I consider the ultimate feature, essential to any editor, the ability to perform macro changes in the text file. This means that if Constantinople is spelled wrong in 87 places throughout the text, a one-line command will go through the file and replace the incorrect spelling with the correct one. Both PIE and VS allow one to look at each occurrence to confirm the change. This is also useful for locating a particular place in a file.

So far, both editors look about the same. They're not, and here's how they differ. With PIE, you will need to become familiar with the manual and the commands, and you may have to refer back to refresh you memory. With VS, there is a very comprehensive "help file" that may be kept on-line on the active (VS source) disk, available to be called up without interrupting the edit session when you forget a command. This takes a fair amount of disk space (57 sectors) but on a two-drive system this is no problem. And since you can replace the disk in a one-disk system when you're ready to write out and save the file, single-drive users can keep their help file available, too.

Another major difference is in the use of the H19/H89 special-function keys across the top of the keyboard. Both PIE and VS use them, but VS uses the CRT's 25th line to display the current meaning assigned to the keys. This is important, because while PIE's functions remain the same throughout the session, VS changes the definitions depending on what you're doing. For instance, keys F1-F5 each have a main function, but when one is pressed, say the Input/Output function key, all five are re-defined into Read, Write, Exit, Insert and Delete. This may not seem terribly clear in reading, but when displayed on the screen the effect is to dramatically simplify the operation of the editor. I never refer to the manual for VS, except to get the address of the manufacturer. And, as mentioned, if the name of the function isn't sufficient to refresh your memory, the HELP function key is available. There's also an "oops" key that undoes the effect of a command typed by mistake (if you haven't gone too far along with it).

VS allows the user to set both the left and right margins, and allows for right justification, or filling, of the text. This is not so satisfactory as on more expensive text processors like AutoScribe or Magic Wand, as the justification all seems to take place on the rightmost five or six words, leaving the left part of the text unchanged. As a result, I rarely use that feature. (It's neat to watch it work on the screen, though.)

Finally, VS will allow you to insert text from another file into the working file at the location of your choice. PIE will not, and while this may not be a crucial feature to you, it is often highly

(Continued on next page)

HARDWARE REVIEW

The Sinclair ZX81

Small Review of a Small Computer

by Jerry Chamkis

Jerry Chamkis, P. O. Box 18093, Austin, TX 78760.

The ZX81 is the recent entry from Sinclair for lowest cost computers, and a very delightful entry it is. It retains the innovative features of the older ZX80 such as single-keystroke keyword entry and a very compact case. The new machine has an 8K ROM that contains the BASIC interpreter, the video display firmware, and a floating-point calculator that offers about the same amount of power as an Apple but at about double the speed. The BASIC interpreter allows variable names of arbitrary length, so you can write programs that are very nearly self-documenting.

The basic system also includes PEEK, POKE, and USER functions to facilitate machine language programming for the Z-80A processor. There is no doubt that the software resident in the system comes from a very smart group . . . they describe themselves in the excellent BASIC programming manual as "a small firm of Cambridge mathematicians."

Sinclair is an English company and there is one English chip in the machine – a large universal logic array which executes the video, cassette tape and address decoder functions. The remaining four of the IC's in my kit are of Japanese manufacture.

The small keyboard is not as hard to use as it might seem and is, in fact, just right for young fingers to practice the marvels of computing. If you should wish to add a larger keyboard, nothing more is required than a naked keyboard wired into the ZX81. About $1 in parts will add a direct video output, so you can use a composite monitor instead of a television set as your display device; another 50¢ worth of parts will fully decode the ROM address space to allow for easy expansion of the control program.

Sinclair and others offer RAM extension boards that start at 16K bytes and $100. Don't take seriously the rather disappointing article in the January issue of *BYTE* . . . it is neither impossible nor undesirable to add more than 16K of RAM to this machine. I am equally pleased with its performance as a BASIC learning machine as a smart process controller, for which I predict it has a glorious future. It costs $150 assembled or $100 in kit form, which includes a UHF TV modulator and a calculator-charger type power converter. All in all, a rare, excellent bargain!

(Editor's note: see our May issue for details of the above-mentioned direct video output modification.)

Software Reviews

(Continued from page 58)

desirable.

So we come to the bottom line: what's to choose between the two? I have owned PIE for over two years, and VS for about one. While I like them both, I prefer and use VS more. The feature I appreciate most is the margin setting, while my wife (who has her own H89) appreciates the help file. VS takes considerably more disk space, especially with the help file installed, and takes a bit longer to start up. With regard to the price, the Software Toolworks' latest version of PIE, 1.5b, sells for $29.95 at most Heath retail stores and direct from the maker, while VS sells for $39.95, also from the Heath stores or the maker. Both are top-quality programs from highly reputable firms. I believe the price difference is due to the development cost; it probably cost more to create VS than PIE.

These are by no means the only good, full-screen editors available to the Heath user; they're just two of the best I have personally experienced. When combined with a text formatting program (like Runoff or Walt Bilofsky's TEXT) comprise a text processing package for about $75.00 that can't be beaten for hundreds of dollars. When used with a daisy-wheel printer, such a combination is fully capable of producing camera-ready copy for offset printing, with headers, footers, page numbering, strike-over, footnotes, full justification, line spacing and so forth. ▶▶J

BOOK REVIEWS

The Logical Design of Multiple-Microprocessor Systems
by B. A. Bowen and R. J. A. Buhr
Prentice-Hall, Inc.
310 pages, $25.00
Reviewed by George W. Jolly

Concurrent programming is the managing of software in which more than one thing happens at the same time. With the current low price of microprocessors, the design of such a system can easily involve assembling modules of software and hardware in a custom configuration to solve a concurrent problem. *Logical Design* takes you through this design process, beginning with the concept of concurrent systems. Whether your needs involve software on large timesharing systems or in clusters of microprocessors, this book will serve as an excellent text.

The concrete examples all assume the Intel 8080 microprocessor, and are usually in PL/M or Pascal. However, the vast bulk of the illustrations are diagrams, not program excerpts. These diagrams are very helpful and are not specific to any particular computer. A packet-switching protocol provides a concrete example of design trade-offs in a working system.

You should be warned that concurrent systems involve advanced concepts. If one hasn't mastered ordinary programming, he will have a bit of fun trying to debug a concurrent program. *Logical Design* assumes plenty of experience in non-concurrent work. One strong point to the book is that several approaches to a given problem are worked out to expose the limitations or failings of each before settling on a better approach. This can save considerable wasted trial-and-error in a first concurrent project.

Logical Design explains concurrent systems in an onion-skin fashion, where an innermost kernel is surrounded by successively higher-level layers of software. Several types of monitor routines are shown, which logically protects access to some resource. Diagrams also explain methods of hardware handshaking between devices. Examples of concurrent software implementation in both PL/M and Pascal show how these languages can be used, even though they were not designed as concurrent languages.

Logical Design is a thorough text on an interesting subject. If you want to learn about concurrent systems in a variety of styles, then *Logical Design* is for you.

Threaded Interpretive Languages
By Ronald G. Loeliger
BYTE Books
251 pages, $18.95
Reviewed by H. T. Gordon

A subtitle for this fascinating work could be "How to Construct Your Personal FORTH-like HLL." It's a unique, detailed description of the deep structure of this type of computer language, and anyone who wants to know *how* it works ought to read it. To understand it, one needs some familiarity with the architecture of microprocessor-based systems and a modicum of skill in assembly-language programming (though not necessarily that of the Z80 used in many of the specific examples). The underlying FORTH concepts are clearly explained in general, but the implementation is "non-standard" and is therefore named ZIP. It's unlikely that many readers will emulate the author and create their own core-language. But the tutorial impact on microcomputer software design may be impressive, influencing new ROM monitors, operating systems, assemblers, and even the HLLs that are not FORTH-like; especially in 6809 systems, since this microprocessor is far more stack-oriented than the Z80.

ZIP looks like a powerful variant of FORTH, stressing code-efficiency so that its core will "fit" into 4K bytes, of which 3.2K are used for the 150-odd keywords (summary on pp. 175-179) and the rest for the outer and inner interpreters. The numerous trade-offs required by this high compaction are analyzed with admirable frankness. Only the first 3 characters of source-code keywords are retained (e.g., both DROP and DROX would have the same 4DRO dictionary header). The severe timing penalty of defining simple operations as : . . . ; "subroutines" instead of direct machine code is emphasized. Many primitives transfer only one byte to and from the parameters stack, instead of the usual 2-byte "word." Although an HLL is a system-independent virtual machine, its I/O must of course interconnect with system-dependent routines (ideally via programmable vector locations). Loeliger says little of this (pp. 54, 74), seeing it as a distinct problem.

The plural "languages" of the title merely indicates that many other variations of the FORTH theme could be constructed. These are not explored. Although he also has UCSD Pascal, he does not hint at possible resemblances of its compiled "pseudo-code" to FORTH-type "threaded code," nor does he refer to the *general* problems of compilation – invariably some way of linking prefabricated blocks of machine code. He does, however, implement a full Z80 assembler and sketch the outlines of an editor and lightly touch on disk-access and many other practical problems. Most commercial FORTH systems require the presence of an elaborate operating system such as CP/M (a massive tail waggable by quite a small dog!).

The book is expensive for its size (30% of its thickness is in its hardcovers). It may be criticized for incompleteness and lack of generality, and its unorthodox approach will displease many users of FORTH. But its primary task – explication of the generalized FORTH virtual machine – is done very well, and it can be a take-off point for exploring the possibilities.

Microsoft BASIC Decoded
& Other Mysteries for the TRS-80
by James Farvour
IJG Computer Services
310 pages, $29.95
Reviewed by James Cardell

With one major caveat, this book is a valuable resource for the experienced assembly language programmer who desires to get the most out of a system which runs Microsoft BASIC. Included in the book is a concise but complete description of the system monitor and BASIC interpreter. Armed with this knowledge of how the BASIC runtime environment works and what functions are available in the interpreter code, and guided by the many how-to-interface-it examples, programmers will achieve quantum increases in their programming abilities.

Manuals of this sort have been available for a few years now and as an attempt to fill the enormous need of documenting Microsoft BASIC and the TRS-80 system monitor (undoubtedly to the chagrin of those in Forth Worth who refuse to acknowledge the existence of anything but BASIC programmers). To their credit these earlier attempts shed much

light on this subject, but Farvour's book is much more than an attempt. It may very well be a definitive work.

In addition to providing an abstract description of the runtime environment, he has taken a disassembled output of the low memory in which the system resides and lucidly commented every line (over 7500!!) of code. Thus, this book serves readers on many levels. It is a reference to the programmer attempting to interface the hardware. It is a guide to achieving powerful, concise, modular code by using the available runtime routines. And it is also a detailed instructional manual for those interested in examining a professional systems program. In short, it is a wise addition to any serious TRS-80 owner's library.

All entry points into the code are documented with interface examples consisting of a few lines of code showing how to call the routine and what results to expect. The entry points are organized by functionality, but cross-references by name and address should also have been provided. In addition to these short examples, some larger programs are also presented. A SORT program is given to demonstrate the technique of adding reserved words to those that the interpreter already recognizes by taking control at an error entry point. An OVERLAY program is an example of a sophisticated system program for a disk system that provides the facility to add statements to a running BASIC program while maintaining current variables.

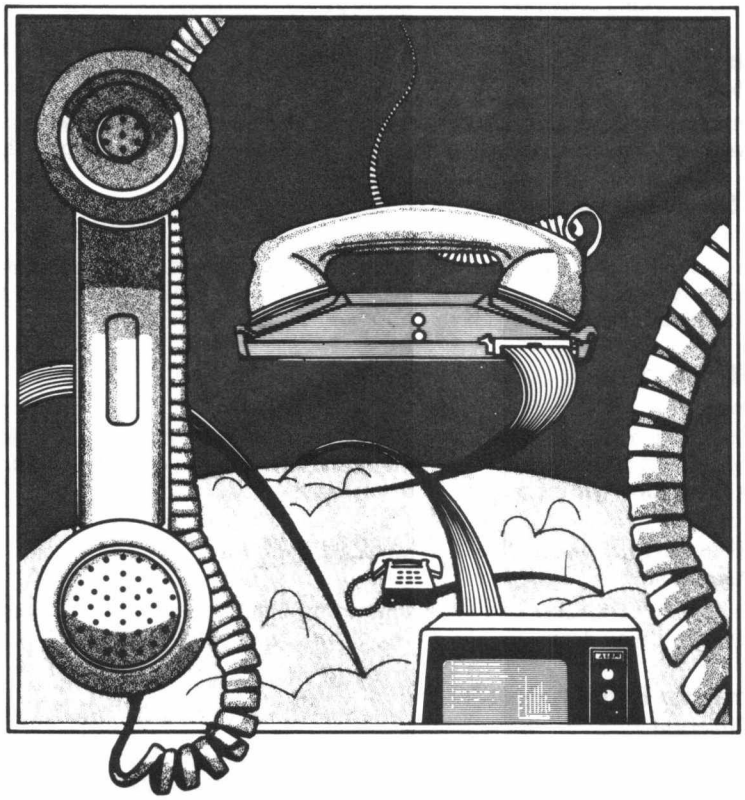
Now as for the caveat, the book is, out of legal necessity, incomplete. Because of copyright restrictions, the disassembled code could not be reproduced in its entirety in the book. The alternative taken here was to excise parts of the code and give instructions to the reader as to how these parts may be reproduced. The "parts excised" is the disassembled instruction operand field. The "reproduction instructions" are to print a disassembled listing of low memory on your own system and collate its pages with those of the book, which is conveniently drilled for a three-ring binder. Of course, to carry out the reproduction you need a line printer. And further, the recommended disassembler which formats its output to align with that of the book is the one provided with Apparat's NEWDOS, so you also need a disk system to do the job right.

For some reason, (be it legal or the desire to sell many books) this information is never completely presented in the book. In fact, it is buried in a few sentences at the beginning of Chapter Seven. This is no problem for those purchasers possessing the requisite system configuration, but there will probably be irate readers with minimal systems unable to complete the book. This situation does not render the book totally useless, rather it makes the examination of disassembled code more cumbersome. It is possible to run a disassembler and list the output on the video monitor. This may then be read along with the comments in the book. The missing code might even be written into the appropriate space in the book, but this is impractical for the entire book. Reliance on the video output eliminates the ability to easily refer to it when using the system for something else, editing an assembly program for example. ▶▶J

DR. DOBB'S JOURNAL
For Users of Small Computer Systems

$2.50
($3.00 in Canada)

Number 67 May 1982



Telecommunications . . .

. . . Computer Bulletin Boards . . .

How to Add a System Clock

John S. James

Jerry Chamkis

Gene Head

more features inside!

DR. DOBB'S JOURNAL
For Users of Small Computer Systems

May 1982 Volume 7, Issue 5

Editor – Marlin Ouverson
Contributing Editors –
Dave Caulkins, Dave Cortesi
Marketing Director – Craig Harper
Marketing Assistant – Larry Brittain
Advertising Sales – Laird Foshay
Circulation – Gloria Romanoff,
Theresa O'Rourke, Terri Marty
Art Director – Clifford West
Production Manager – Barbara Ruzgerian
Production Assistant – Joneva Barry
Typesetter – Renny Wiggins
Proofreader – Nancy Heubach
Cover Illustration – Al McCahon

© 1982 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $40 for first class to Canada and Mexico; $58 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Thomas H. Grohne, G. V. Elkins, Robert M. Connors, William J. McEown, Robert C. Luckey, Robert N. Stabler, DeWitt S. Brown, John B. Palmer, Jack Miller, Burks A. Smith, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Howard E. Decker, Jim Wilson, Frank Lawyer, Chris Pettus, Paul Lutus, Ben Goldfarb, Rodney Black, Steven Weiss, Pat Phelan, John Brodie, James Fleming, Unison World, Christopher Sherbak, Thomas Davis. **Sustaining Subscribers:** Gordon Eubanks, Jr., Pete Roberts, Steven Fisher Enterprizes. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Hofacker-Verlag, Tegernseer Strass 18, D-8150 Holzkirchen, West Germany. **Asia & Australia:** ASCII Publishing Co., Ltd, Aoyama Building 5F, 5-16-1 Minami Aoyama, Minato-ku, Tokyo 107, Japan.

CONTENTS

18 Program for Modem: Forth Code From the Conference Tree
by John S. James
Programming a modem can be one of the trickier aspects of a telecommunications system. The article explains techniques used by the CommuniTree Group, and includes actual working code now placed in the public domain.

26 North Star-to-Epson Connection: A Simpler Way
by Larry Shoer
DDJ has published articles and software in the past about using this popular parallel printer with a North Star. Author Shoer has come up with a simpler approach.

30 Adding a System Clock
by Michael Kan
Despite the general usefulness of system clocks, many computer hobbyists do not use them. This feature shows how to implement a software-simulated clock in an independent processor external to the host.

42 New USER for CP/M
by Gene Head
Anyone who does a lot of hard copy documentation may have found hand feeding single sheets to present a thorn in the side. Perhaps single sheet ejects and an operator controlled waiting period are the answer for you.

48 Direct Video Output From the Sinclair ZX81
by Jerry Chamkis
This author tantalized Sinclair aficionados last month with his mention of using a a monitor as the display device for the ZX81. Read on, for jitter-free video!

50 Using Computer Bulletin Boards
by Marlin Ouverson
A pack of people plead monthly for the how-to's of computer conferencing and bulletin boards. Here lies a logon sequence to get you onto most CBBS's.

4 Letters
4 Editorial
10 Dr. Dobb's Clinic
13 Of Interest
53 Software Review
A TRS-80 Smart Terminal Package
56 Book Reviews
58 Hardware Review
Hayes' Stack Smartmodem
59 Guest Essay
Prior Censorship

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, Box E, 1263 El Camino Real, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025 · 415/323-3111

LETTERS

Screen-Oriented Bugs

Dear Marlin,

I want to thank you and all the people at *Dr. Dobb's Journal* for publishing my article, "A Portable, Screen-Oriented Editor" (*DDJ* #63).

Your readers should know about several bugs and typos that have been found in that article. Mr. David Harper of Henrietta, New York gets my nomination as "exterminator of the month" for his work in finding bugs 1 through 8 below. So far, he is the *only* person who has written to tell me about any bugs in the editor.

Bug 1 In the function search1() on file ed3.c the code that reads:

```
if (pat[0] == EOS) {
    return;
}
```

should be changed to:

```
if (pat[0] == EOS) {
        return -1;
}
```

This bug has a small effect on the find and search command-mode commands.

Bug 2 In the function pmtfile() on file ed7.c, change the line:

```
pmtcol1();
```

to:

```
pmtcol1(x);
```

This bug causes the prompt line to contain an erroneous column number following the load or name command-mode commands.

Bugs 3, 4, 5, and 6 I managed to botch the routine syscstat() on file ed8.c. The version of this routine for CP/M 2.2 which was published in *Dr. Dobb's Journal* does work, but it doesn't do what the comment says it should. In addition, the version of syscstat() for CP/M 1.4 contains a separate bug. In the interest of clarity, I won't print the erroneous versions of syscstat() here.

The correct version of syscstat() for CP/M 2.2 is:

```
/* return -1 if no character is ready
   from the console.
 * otherwise, return the character.
 */
syscstat()
{
    int c;
```

Editorial

Electronic Graffiti

Telecommunications is one of the most important uses for microcomputers. It can mean many things: locally linked machines in a separate-but-equal file sharing mode, or in a master-slave(s) relationship; hooking in to centrally controlled data bases; or networking remote micros as independent nodes, over telephone lines or through the airwaves. But it can mean only two things: power to control information, and power accrued by access to information.

Commercial data bases like the Source, MicroNet, and DIALOG are among the telecommunications schemes which get the most press coverage, thanks to big financial interests. Decisions about which information to provide, how easily, at what price (usually between $5 and $100 per access hour), and to which audiences, are effectively made in the corporate boardroom.

Local networks like Nestar, Ethernet and others grow out of "futuristic" office concepts. These are commercial services which link computer workstations under one roof. They allow inter-communication; shared access of files on, for example, hard disk; and monitoring of each operator's actions, especially useful in supervisory or educational environments. Local networks are installed at prices which are low enough, relative to the computing power created, for most financially stable institutions.

Of growing interest, and especially important for computer hobbyists, are the hundreds of no-charge Computer Bulletin Board Systems (CBBS is actually a trademarked acronym). Around the world, microcomputer enthusiasts are putting up CBBS's as a public service, for the interaction with others, and for the experience. They use telephone lines and their own one- and two-disk drive configurations to link users in active interchanges, to provide public notices, articles, and conferences. Some are operated for use by special interest groups in the sciences, computers, arts and politics.

The importance of such homegrown networks should in no way be underestimated. They are quickly mastered, thanks to their ease of operation. Because they are free of charge, there is no stratification of users by economic range. An unwritten operator's code of ethics severely restricts censorship or "editing" of messages, creating a forum for sometimes radical, sometimes ingenious thought.

The foundation is already laid for scenarios from a science author's imagination. At night, humans go to bed as their machines blink quietly, passing messages from one to the next. Disk drives store personal mail and any unprotected through-messages with embedded, pre-selected keywords. A chain of local transmissions travels electronically across several states, avoiding long-distance fees. Broadcast repeaters transmit cross- or inter-continental files, with stringent error checking at each node. CBBS's unite periodically to share selected resources, giving each local user indirect access to an entire world of information.

In our Information Age, those who control information are the ones with real power – personal and political. Given such a scenario, microcomputer-controlled message systems are like local power generators; anyone with a terminal or a micro can use the resultant energy free of charge.

Marlin Ouverson

Editor

```
        c = cpm(6,-1);
        if (c==0) {
                return -1;
        }
        else {
                return c;
        }
}
```

The correct version for syscstat() for CP/M 1.4 is:

```
syscstat()
{
    if (bios(2,0)==0) {
            return -1;
    }
    else {
            return syscin();
    }
}
```

Please note that I still have no way to test the CP/M 1.4 version of syscstat(). Proceed with caution.

Warning And again I say, *warning*. If you make these changes to syscstat(), then you must also change the code in chkkey() on file ed3.c. The old code is:

```
if (c == 0) {
    /* no character at keyboard */
    return(NO);
}
```

The code should be:

```
if (c == -1) {
    /* no character at keyboard */
    return(NO);
}
```

Bug 7 In the function bufdeln() on file ed10.c the code that reads:

```
if (bufdn()==ERR) {
    bufline=oldline;
    oldbufp=bufp;
    return(ERR);
}
```

should be changed to:

```
if (bufdn()==ERR) {
    bufline=oldline;
    bufp=oldbufp;   /* bug fix */
    return(ERR);
}
```

This bug fouls up error recovery.

Bug 8 The following line of bufrepl() should be deleted:

```
bufpmax=bufpmax+n-oldlen;
```

The line already appears in the routine bufext(). This bug makes parts of the main memory buffer unusable as editing progresses. Do not change the bufext() routine!

Bug 9 I did manage to find one bug

Figure 1
Diskdoc Changes

```
/*
 *  get character, no echo
 *  upper case is converted to lower
 */

conlower()
{   char ch;
    ch=conin();
    if ((ch>='A')&(ch<='Z')) ch=ch+'a'-'A';
    return ch;
}


/*
 *  select drive, drive name is 'a','b' etc.
 *  set values for track, sector counts and first sector
 *  return true if ok
 *  to find out what this version does,
 *  refer to the cp/m 2.0 alteration guide
 */

seldrv(drv,pt,ps,pf)
    char drv;
    int *pt,*ps,*pf;        /* where to put track, sector and firstsector */
{   int *dph,*spt,*dsm,*off,halfsecs,trks;
    char *dpb,*bls;
    if ((dph=bios(9,drv-'a'))==0) return 0;
    xlt=dph[0];             /* look at disk parameter header */
    dpb=dph[5];
    if (xlt) *pf=xlt[0];
    else *pf=255&bios(16,0);
    spt=&dpb[0]; *ps=*spt;  /* and at disk parameter block too */
    bls=&dpb[2];
    dsm=&dpb[5];
    off=&dpb[13];
    /* this is tricky since unsigned divide isn't supported */
    halfsecs=(*dsm+1)<<(*bls-1);
    trks=((halfsecs/(*spt))*2)+(((halfsecs%(*spt))+(*spt-1))/(*spt));
    *pt=trks+*off;
    return 1;
}


/*
 *  bios call
 *  will only work with the original small C
 *  other C compilers will usually have the argument sequence reversed
 */

bios(fun,arg)
    int fun,arg;
{   char *ofs;
    ofs=(fun-1)*3;
#asm
    pop     d           ;ofs
    pop     h           ;ret
    pop     b           ;arg
    push    b
    push    h
    push    d
    lhld    1           ;get pointer to bios
    dad     d           ;add offset
    lxi     d,retn1
    push    d
    mov     d,b         ;arg in de too
    mov     e,c
    pchl                ;go
retn1:
    xchg
    mov     l,a
    mvi     h,0
    pop     b           ;ofs
    push    b
    mov     a,c
    cpi     (9-1)*3     ;select disk function?
    jz      retn2
    cpi     (16-1)*3    ;sector translate?
    jnz     retn3
retn2:
    xchg                ;value came in hl
retn3:
#endasm
}
```

on my own. In the routine edjoin() on file ed4.c change the lines:

```
/* abort if the screen isn't wide e-
   nough */
if (k>SCRNW1) {
    return;
}
```

to:

```
/* abort if the screen isn't wide e-
   nough */
if (k>SCRNW1) {
    bufgetln(editbuf, MAXLEN);
    return;
}
```

The effect of this bug is striking when it happens and you should definitely fix it. You can see the effect of the bug as follows:

1. Use the join key to join two lines whose combined length exceeds the width of the screen. Nothing (apparently) will happen.
2. Move the cursor to the middle of the current line.
3. Make some change to the line from edit mode. You will now see that the line you are editing isn't the line that you should be editing.

There are also several typos in the article that may be confusing:

Typo 1 On page 20 of the article, in the paragraph about the split key, it says,

> "Both these keys also switch the editor to insert mode."

This sentence should be deleted. Somehow it was duplicated from a previous paragraph which concerned the insert up and insert down keys.

Typo 2 In Figure 4 of the article, the function of the s command should be changed from:

> move cursor right to H char K

to:

> move cursor right to <char>

Typo 3 The comment for edattop() on file ed4.c should be changed from:

```
/* return true if the current edit
   line is being
 * displayed on the bottom of the
   screen.
 */
```

to:

```
/* return true if the current edit
   line is being
 * displayed on the top of the
   screen.
 */
```

Yours,
Edward K. Ream
1850 Summit Ave.
Madison, WI 53705

Doctoring Diskdoc

Dear Marlin Ouverson,

Since I submitted the Diskdoc listing (*DDJ* #66), two bugs have been found. The first one concerns compatability with other compilers. While Small-C evaluates expressions from left to right, the Kernighan/Ritchie C-language specification leaves the order of evaluation unspecified. The previous version of the *conlower()*-function depended on the order of evaluation, the version shown as Figure 1 does not.

The second bug was caused by an oversight of mine when interpreting the CP/M BIOS disk parameter information. On some double-density formats (mainly the Intellec MDS II), Diskdoc believed that the first sector was number zero, instead of number one. The enclosed revised versions of *seldrv()* and *bios()* will perform correctly.

Yours sincerely,
Egil Kvaleberg
Studpost 111
N-7034 Trondheim-NTH
Norway

Sad About Sinclair

I am an owner of a Sinclair, ZX81 computer and I am dissatisfied with its operation. I experience frequent, unpre-

Figure 2
Small-C Shrinker

```
/*                                      */
/*      Declare local variables         */
/*      (i.e. define for use)           */
/*                                      */
/* works just like "declglb" but modifies machine stack */
/*      and adds symbol table entry with appropriate */
/*      stack offset to find it again                */
declloc(typ)            /* typ is cchar or cint */
        int typ;
        {
        int k, j, newsp; char sname[namesize];
        newsp=Zsp;
        while(1) {
                if(endst())break;
                if(match("*"))
                        j=pointer;
                        else j=variable;
                if (symname(sname)==0)
                        illname();
                if(findloc(sname))
                        multidef(sname);
                if (match("["))
                        {k=needsub();
                        if(k)
                                {j=array;
                                if(typ==cint)k=k+k;}
                        else
                                {j=pointer;
                                k=2;}
                        }
                else
                        if((typ==cchar)
                                &(j!=pointer))
                                k=1;else k=2;
                newsp=newsp-k ;
                addloc(sname, j, typ, newsp);
                if (match(",")==0) break;
                }
        /* change machine stack */
        Zsp=modstk(newsp) ;
        }
```

dictable system failures due to the power jack plug (the "on/off switch") connection, keyboard depressions and the edge connector contact for the 16K RAM pack. The system failures result in loss of time and effort in re-entering and re-running programs. My ZX81 sits on a shelf instead of being used because of the likelihood of its error prone operation.

I would like to hear from other owners of the ZX81 who are experiencing these same failures and who are interested in sending a group message to Sinclair requesting a fix. Contact me at the address below. I will respond to your replies through the same publications which have published this message.

Donald F. Shank
POB 50301
Palo Alto, CA 94303

Editor's note: For the other side of the coin about Sinclair's latest computer, see last month's DDJ *Hardware Review and "Direct Video Output" in this issue.*

I Can C Forever

To the good doctor:

Having read the glowing reports in *DDJ* about Small-C, I ordered the CP/M (tm) version from the CODE WORKS. First I would like to say that Ron Cain has provided a true service to computer hobbyists by placing this piece of software in the public domain. Also, the CODE WORKS is performing a service to hobbyists using CP/M systems by providing this compiler at the very reasonable price of seventeen dollars.

One very easy way to implement code improvement to the compiler is in the routine for the declaration of local variables "decIloc(typ)". I have changed this routine so it will only modify the stack pointer after the entire declaration has been parsed. Before, the stack was modified for each variable in the declaration. The simplest statement to benefit from this is:

```
char a,b ;
```

This now generates a PUSH rather than 2 DCX SP's. For a statement like:

```
int a1(40),a2(40),a3(40),a4(40),
    a5(40) ;
```

the savings in space and time are considerable.

I was not expecting the compiler to benefit from the changes, however it did (although not by much). In the process of bootstrapping this change, the code size changed by the following amount:

13	bytes larger: new compiler compiled with old compiler
121	bytes smaller: new compiler benefitting from compiler change
108	bytes smaller: net change in size.

I am enclosing a copy of "decIloc" for those wanting to duplicate this code improvement (see Figure 2 on page 7).

Sincerely,
Dan Bates
Rt. 8, Box 278
Johnson City, TN 37601

Pidgin Tracks

Dear Doc:

It looks as though a slight plaque buildup has occurred on the proofreader's glasses, which manifested itself in *DDJ* #65. In the Editorial, it is stated that "as Heinlein noted in his great Foundation trilogy, . . ." My copies were authored by Asimov. Heinlein's forte is more in the area of serendipity.

In a more serious vein, I really appreciated the PIDGIN follow-up, and read and analyzed its approach with great interest. It appears from the article that a listing of PIDMATH.ASM, which includes arithmetic routines ?ISUB, ?IMUL, and ?IDIV, among others was omitted. See page 28, rightmost column, seventh trhough ninth and 33rd throught 38th lines. I hope that you will run these as an errata in an upcoming issue.

Personally, I have been implementing TINCMP and PIDGIN on my SOL-20 by a method of successive approximation, as follows:

(1) Verbatim entry of W. A. Gale's TINCMP,
(2) Revision of (1) to wholly structured code without GOTO/LOC construct,
(3) Translation of a complete pattern set from 6502ML to 8080AL using "Template Definition" coding form, and TINCMP syntax summary,
(4) One-for-one replacement of (2) statements with (3) code as a purely mechanical process, using my text editor and my fingers and eyes as a TINCMP surrogate.

Presently, I have only proceeded to this point. As the SOLOS operating system of my tape cassette-based machine is markedly different from CP/M and Gale's basic disk operating systems, the going is slow.

As my approach to an 8080 implementation differs due to changes in operating system and programming style (I formerly ran a service bureau in Washington, DC, and went for two and a half years without writing a single "GOTO"), I will let you know of my progress.

If you can convince Gale to send you his "seven Pidgin programs which test the Pidgin macro definitions fairly thoroughly" (*DDJ* #65, page 25), and fuller descriptions of Maclod and Spartan, I am certain that they would be popular features.

Keep up the good work. And watch out for those gremlins.

Very truly yours,
Russell P. Herrold III
2834 Clifton Road
Columbus, OH 43221

Foundation Folly

To Marlin Ouverson:

I'm sure you will get many letters pointing out your glaring editorial error in your March issue of *DDJ*.

I refer to your comment that Heinlein wrote The Foundation trilogy. Although a great author, I'm sure Issac Asimov does not want to lose credit for his greatest work.

As a charter member of the Science Fiction Book Club back in 1960 my first book was the trilogy & after 20+ years I still have it – beat up – well read by many – but still the best.

Frederick G. Volpicelli
331 8th Ave.
Pelham, NY 10803

Dear Mr. Ouverson,

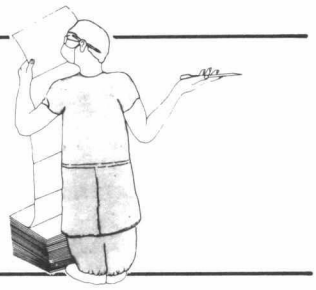
In response to your editorial in the March issue which you conclude by asking "I wonder what'll happen next month?" . . .

I think I know. You will probably get about 400 letters reminding you that Asimov, not Heinlein, wrote the Foundation trilogy.

Sincerely,
Michael W. Brown
2212 Cloverleaf Place
Ardmore, OK 73401

Editor's note: Thanks to those who (too eagerly, methinks) caught the gaffe. I'm sure Mr. Heinlein would've written "Foundation" if he'd thought of it . . . ■■

DR. DOBB'S CLINIC



by D. E. Cortesi

A CPU Transplant

Bruce Komusin of Huntington, NY, sent us a brief description of two interesting pieces of work. First, he performed a brain transplant on his 8085-based system:

> "The 8085, running from a 10 MHz crystal, was removed and replaced with a small card containing an 8088, an 8284, a sprinkle of resistors and caps, and a 14.58 MHz crystal (giving a 206ns clock). All other components of the system were unchanged, including 64K dynamic RAM, memory-mapped video, etc."

We'd no idea that an 8088 would slip in so easily. Of course, Komusin then had the problem of running all his existing software. He elected to write an 8080 emulator in 8088 machine language, and . . .

> "I must say the results have been underwhelming. The emulator ran 5.85 times slower than the 8085 when executing a compute-bound RAM test. When it executed an I/O-bound full-screen editor, the response was acceptable but the prompts and error messages, as well as the scrolling, were noticeably slower.
>
> "The emulator itself fits in one 2732 EPROM which resides outside the first 64K of address space. Interrupts are not implemented. The mechanism used to emulate the 8080 instructions could be described as Direct Threaded Code. The "inner interpreter," to borrow a FORTH term, is three bytes long; it consists of two 8088 instructions, neither of which affects the condition flags. The 8088 can emulate all but two of the 8080 instructions (IN and OUT) in thirteen bytes of code or less."

If you'd like to know more about Komusin's CPU transplant, bombard us with letters. We'll pass them on, and maybe he'll write us an article.

Getting WiDIR

Is anyone still using CP/M 1.4? You are? Are you tired of having the DIR command's output roll off the top of the screen? Dusty Pulver, who lives in Ampthill, Bedford, U.K., got tired of it, and decided to make DIR give a wider display. Pulver describes the patch:

> "It formats each file name into a block 16 bytes long, and doesn't output any CR or LF, so it will work on any screen that's a multiple of 16 columns wide, i.e., it puts four names across a 64-column screen and five on an 80-column screen. However, output to a printer may cause problems – so use STAT for that.
>
> "After the names have been printed, the system performs a warm boot, because I've had to use the routine that outputs 'Bdos Err on x:,' which then does a boot, and I can't find a way around it. Also, due to space restrictions I had to use a relative jump, so the patch will work only on a Z80 system. Finally, if there's a file with a filetype of three blanks, the format will be messed up.
>
> "Here is the patch as it would appear in a 16K system. You have to add an offset to the indicated instructions for a larger system. The offset is the size of the system less 4000h bytes, i.e., 8000h in a 48K system.

```
0DFD  CD B0 2A   call 2AB0h+offset
0E00  C6 41      add  A,41h
0E02  32 8F 2D   ld   (2D8Fh+offset),A
0E05  01 8D 2D   ld   bc,2D8D+offset
0E08  CD 5D 32   call 325D+offset
0E0B  18 05      jr   05h
0E0D  20 20 00   db   blank,blank,0
0E10  3A 24      db   colon,dollar
```

The Eccentric Type

Here's an item from the Pitfalls and Warnings file, Hardware Section. Our Diablo 1650 printer (an excellent machine) recently broke a spoke on its typewheel – the "y" flew off. "Why not," we mused, "and at the worst possible time, too." We had 150 pages of camera-ready copy to print. So we hustled out to pick a a new Titan 10 daisy.

The new wheel was quickly installed, but there was something fishy about the output. Nothing specific, it just looked messy somehow. Examination with a magnifying glass revealed the problem: the letters weren't printing on an even baseline. Some sat high, some low. It was only a matter of 64th of an inch, but the eye can detect a very small error in alignment.

Back we went to our suppliers, who stood by patiently while we tested all the Titan 10 wheels they had in stock. They all printed a wavy line. We don't know what the problem was. The spokes might have been different lengths, but more likely the axlehole was not concentric with the ring of letters. Whatever, Diablo's quality control had slipped.

Fortunately, our dealer also carried Diablo-compatible typewheels from another company, AGT. The AGT version of the Titan 10 worked perfectly, and so we managed to meet another deadline. Has anyone else had a problem with an eccentric daisy? What was the outcome?

Bye-bye Files

From Pitfalls and Warnings, Software Section, CP/M department: We've been building a series of CP/M utilities. They all use the same command syntax, namely

program *input.typ* [*output.typ*]

Now, the *output.typ* operand can name the same file as *input.typ*. In fact, if *output.typ* is omitted, the programs assume that the output file is to replace t'.e input file – but not, of course, before the program is finished.

All the programs use the same code to set up the files. It does several things:

(1) open *input.typ*,
(2) erase any file *output*.$$$,
(3) create a file *output*.$$$.

The program then writes to *output*.$$$. At the end of the run, it

(4) erases any file *output.typ*,
(5) and renames output.$$$ to output.typ.

The procedure worked just fine until, in a fit of stupidity, we ran one of the programs with an omitted *output.typ* and an *input.typ* that was an ambiguous file reference. Not "*.*," fortunately, but "*.OUT," and we had several files with the .OUT filetype. The events that followed were absolutely correct, given the file print of the CP/M documentation, but they were unexpected even so. The operand "*.OUT" became

"????????.OUT"

in the default FCB in low storage. The program then

(1) requested an open for
"????????.OUT"

which worked perfectly. It got the first file that matched that reference. Then it

(2) did an erase of "????????.$$$" – no problem here, either. Then it

(3) "made" the file "????????.$$$." That, too, worked fine. It created a directory entry for a file of that name, and the program wrote its output under that name. Things got more dramatic when the program

(4) did an erase for "????????.OUT" and neatly removed a half-dozen files. Following that, it

(5) successfully renamed

"????????.$$$"

to "????????.OUT" and, having illustrated the old saying about how the operation was a success but the patient had died, terminated normally.

There is now a check for ambiguous file references in our standard code for file-handling. Is there in yours?

Department of Amplification

Jim Howell of San Jose, CA, wants to correct one of our careless statements:

> "In the January 1982 Clinic you state that MP/M is the smallest multi-tasking operating system you know of. I would like to point out the existence of OS-9 for the 6809, available from Microware Systems Corporation (5835 Grand, Des Moines, IA 50312). Microware recommends a minimum of 32K bytes of storage for running OS-9 level 1. OS-9 is available for most SS-50 systems that use a 6809 processor."

According to the ad that Howell enclosed, OS-9 supports re-entrant command code so that more than one user can run a single copy of a program, and it allows any command to be started for independent execution while the user carries on with another one.

Submitting Is Not Giving In

Jim Shaw of Indianapolis, IN, has a problem. He wants to drive the ED editor from a script of commands submitted to it via XSUB. That's a sensible idea; the combination of ED and XSUB is a powerful tool for manipulating files. Shaw's problem is that he can't see how to do it:

> "The Substitute command uses the control-z character for string delineation. When I tried to include the control-z in the command line of the input file, ED thought I wanted to stop entering text! There was just no way to include a control-z in the file produced by ED. My next step was to use a dummy character for the control-z, and use DDT to change the file. When I tried that, SUBMIT reported an illegal control character in the input file. How can I put commands that require control-z into submitted files?"

PIP uses control-z as well, to delimit the string arguments of the Start and Quit options (anyone who doesn't know what they do should check them out; they're quite useful). Getting control-z into a submitted file is really easy, or would be except for a small bug in SUBMIT.

The SUBMIT command should accept the two-character sequence "^Z" (that's two ordinary ASCII bytes, "^" followed by "Z") in its input. It is supposed to translate that two-byte sequence into a single control character, control-z. Any of the 31 control characters can be submitted that way; "^C" should put a control-c in the file, "^[" should result in an ESCape, and so on.

We say "should" and "is supposed to" because there is a bug in SUBMIT for CP/M 2.2. It tests the second character of the sequence incorrectly and rejects it. You will find that the sequence "^z" and "^c" will work but "^Z" or "^C" will not. The patch for this problem is to load SUBMIT.COM under DDT and list the instructions at 0441h. You should see

```
0441 SUI 61
0443 STA 0E7D
```

Change the operand of SUI from 61h to 41h and save a new version of SUBMIT. It should accept "^Z" and other sequences for control characters. You can then submit a script of ED (or PIP, or DDT) subcommands through XSUB with no trouble.

Well, *almost* no trouble. Two more SUBMIT problems will cramp your style. One is a bug: SUBMIT absolutely cannot handle a submitted file that contains a null line. A line that consists solely of CR, LF makes it crash or go catatonic. That's too bad, because you'd like to submit a null line at the end of a list of PIP subcommands. There's no fix for this one, but maybe you can work around it by submitting a control-c instead.

When driving ED, you might like to submit a line containing lowercase letters. Forget it; SUBMIT translates its input to uppercase (anyone know a patch for that?). Even so, there's a lot you can do with submitted ED (PIP, DDT) scripts. And examples of that kind of thing would make good Clinic items.

And, on the matter of Clinic items, we still have a very handsome, plastic reference card for the 6502 CPU from MicroLogic. The first sender of a 6502-related "sponse" gets it.

DDJ

OF INTEREST

by Michael Wiesenberg

Games Galore

What? A computer game based on the stock market? The **Avalon Hill Game Company** sent us news about Computer Stocks & Bonds, a version of their educational Stocks & Bonds board game for "all the major home computer systems." Up to four people can play against each other, or one player can challenge the computer. "Now it's possible," says Avalon, "for one to lose his shirt and walk away smiling – knowing it only cost him $20 to find out what an awful speculator he really is." Software is available on diskette for Apple II, Atari 800, and TRS-80 Models I and III (level II), and on cassette for Pet, CBM, TRS-80 Models I and III, Atari 400 and 800, and Apple II. (Reader Service **No. 402**)

Avalon Hill also offers Dneiper River Line for the same machines, a fictionalized World War II engagement between the Russians and German forces in the southern Ukraine. It is based on the actual Soviet thrust over the Dnieper in late 1943. History buffs are challenged to take the role of the Germans in repulsing Russian efforts to breach the critical defensive position. Avalon Hill gives you game components (actual "men"), representing infantry, artillery, armored units, to place on a color mounted mapboard so you can plot moves as they unfold on your screen. They claim the software is worthless without this additional "hardware," making the game piracy-free. $25 on cassette, and $30 for diskette. The company suggests writing first for memory and program specifics. (Reader Service **No. 412**)

Atari, Inc., and DC Comics, Inc., jointly announced the pairing of electronic home video games and comic books. (Well, I can see some similarities.) This summer, certain Atari video game cartridges will contain comic books featuring characters from the games. Defender is to come in June, with "defenseless humanoids kidnapped by aliens from the earthly city and only the Universal Ship Defender can protect them. The aliens are a dastardly lot – Landers, Mutants, Bombers, Pods, Swarmers and Baiters. The Defender must ward off these no-goodniks, firing laser missiles and detonating smart bombs. If Landers abduct humanoids, the heroic Defender must blast them from the sky, catch the falling humanoid and carry him safely to the earthly city." (Please don't think *I* wrote that deathless prose!) $37.95. Berserk, coming in August, offers "a courageous humanoid dodging through a maze shooting nasty robots before they zap him. Out of nowhere appears grinning bouncing Evil Otto who loves to squash careless humanoids." $31.95. And in September, Star Raiders, in a "galactic shoot-em-up . . . with the television your space ship. Armed with deadly photon torpedoes and protective shield, you steer through distant galaxies searching for alien space craft. A special radar scope lets you anticipate enemy maneuvers." $37.95. All that and comics too! (Do you think it's a coincidence that Atari and DC are both subsidiaries of Warner Communications, Inc.?) (Reader Service **No. 422**)

While we're on about Atari, **EduSoft** has announced the availability of their educational programs written originally for TRS-80 and Apple II on Atari 400/800. Used, they say, in "thousands of classrooms across the country," these programs "feature mathematics instruction, drill and games, and include concepts used in graphing, algebra and arithmetic, and computer simulation." Well, this may be a good way to get the kids to learn something besides lightning reflexes on your home computer. Guess the Rule (recognize relative sizes of numbers) and Reverse (strategy) are in Package W-2. Plot (display and compare graphs) and a different Guess the Rule (algebraic guessing game in arcade format) in Package W-3. Simulated Computer (simulation of a small machine-language computer) in W-4. Fraction Recognition and Mixed Number Recognition in W-5. Cassette is $14.95; disk, $19.95. Add $2 per order for shipping and handling. Write for free catalog. (Reader Service **No. 432**)

Fine Freebies

After playing Computer Stocks & Bonds, you may want inside information on which stocks to invest in. For a free report on high technology companies and their performance in 1981 write to **High Technology Growth Stocks** (a newsletter; don't be surprised if they send you a subscription pitch with their list). (402 Border Rd., Concord, MA 01742; 617/369-5382)

Get a free reference sheet for the TRS-80, giving all the Z-80 mnemonics for opcodes 0-255 in decimal, hex, and octal, ASCII code identification of printer control codes, keyboard characters, graphics codes, space compression (TAB) codes, BASIC's reserved word token codes, four number base conversion tables (powers of 2, 16, 2-to-16 conversion, hex-to-decimal), a program that converts LSB-MSB to hex, and articles that clearly explain conversions in both directions. I've got one; it's packed with information. Send a #10 SASE to **ABS Suppliers**, and specify Model I, II, or III. (3352 Chelsea Circle, Ann Arbor, MI 48104; 313/971-1404)

CAVRI Systems Ink (really) is a free newsletter for interactive video users. CAVRI systems link video recorders and computers. (From the photos in their newsletter, it looks like they use Apples.) CAVRI systems are used to train helicopter pilots, by the Navy for engineering courses, by universities to develop interactive video courses, etc. (26 Trumbull St., New Haven, CT 06511; 203/562-4999)

Applications Areas

We received notice of some applications software for your HP-85. LIST/85 is a mail list program by **Tensegrity, Inc.** It has a zip sort, random access search, and the ability to concatenate records in memory with any LIST/85 disc or tape data file without bringing that file into memory. Each complete address is a record of five fields; each field can have up to 32 characters. Up to 100 records (500 fields) can be in memory at a time. $145 includes data cartridge, disk, manual, holder, and 600 self-adhesive label samples. (Reader Service **No. 442**)

How about a utility to duplicate BASIC or system tapes for your TRS-80 Model I or III? How about paying only $11.95 ($9.95 plus $2 shipping and handling) for it? On a Model III it also converts 500 baud to 1500 and vice versa. You could change Scripsit to 1500 for faster loading, for example. It's called COPY-TAPE, and it can also scan through memory to see the ASCII of the program you duplicated, and has a verify command. Use it on a Model I level II or Model III BASIC (both with 16K or more). **David Modney** guarantees his tape to load, or he'll replace it. He ships within three working days.

Here's another inexpensive program for your TRS-80 from David Modney,

called Automatic Graphing of Functions. It plots formulas in the form Y=f(X) or Y=Y+f(X), graphs discontinuous functions, and plots simultaneous equations on one graph. The graph goes to a printer with just "the push of a key." Modney claims his program is "Bug-Free!" It runs on the same machines as COPY-TAPE, and costs $19.95 plus $2 (s and h). (Reader Service **No. 452**)

I can't resist cute names, and who can resist a video text editor for $90? MR EDit doesn't sound like it's exactly a word processor, but it seems close. It's fully screen oriented. The cursor moves by character, word, line and screen. Deletion is by character, word, or line in any direction. MR EDit has extensive search and replace. It has something called "paste buffer manipulation commands." (Could that be for text movement?) User-defined command keys allow custom configuration to any non-memory-mapped CRT. **Micro Resources Corporation** sells MR EDit for CP/M and MP/M systems with 20K transient program area and a CRT with at least 12 lines of 64 or more characters. It's available on 8-inch single density or 5¼-inch soft-sectored floppy. The manual alone is $12.50; that price is refundable on order of the editor. (Reader Service **No. 462**)

Apple V.I.P. II from **Micro-SPARC, Inc.**, outputs VisiCalc grids to a printer the way they are formatted on the screen, in columns and rows. To customize spreadsheet reference reports, you can select from seven different combinations of column width and number of columns. A fast machine-language disk read and sort routine does all this. It automatically compensates for long formulas by segmenting and printing them in clear continuous blocks, and does the same for very wide reports. It sells for $29.95. (But they forgot to tell us what form it comes in!) (Reader Service **No. 482**)

Computer Camps

Want to get the kiddies off your computer so you can use it, and still keep them happy? Send them to a computer camp! That seems to be the newest wrinkle in specialized camps. There are tennis camps, art camps, music camps, and now **Specialty Camps, Inc.**, will conduct Atari Computer Camps for 10- to 18-year-olds at four locations, two four-week sessions at each. The camps will be at the University of San Diego, Asheville School, in Asheville, NC, Lakeland College in Sheboygan, WI, and E. Stroudsburg State College in E. Stroudsburg, PA. The curriculum has been designed by Atari's education specialists. (Reader Service **No. 492**)

For 12- to 18-year-olds who want to have a week of intensive BASIC with one computer for each two campers, **Lake Forest College**, 30 miles north of Chicago, will have four computer camp sessions from June 20 to August 6. The camp director is Dr. Lowell Carmony, Associate Professor of Mathematics and Computer Studies. (Reader Service **No. 502**)

Computer Camps, at Santa Barbara and Lake Tahoe, offer two-week sessions for 100 10- to 18-year-old campers, and include such computer-related crafts as robot assembly. The counselors also insist that the kids see the light of day. Cost is $795. (Reader Service **No. 512**)

Rancho Oso Summer Camp of Santa Barbara has been around since 1960. They offer 56 9- to 15-year-olds two-week sessions for $675. They recently acquired a bank of Atari 800s, and also stress horsemanship and riding. (Reader Service **No. 522**)

Community Calendar

Graphics Interface '82 will be held in Ontario May 17-20. You're too late for early registration, so it will cost anywhere from $75 for conference students only to $345 for general admission to all tutorials as well. It's in coordination with the University of Toronto, and sponsored by the National Computer Graphics Association of Canada (NCGA) and the Canadian Man-Computer Communications Society (CMCCS). Tutorial topics include computer graphics, digital image processing, raster graphics, computer-aided design, graphic design for computer graphics, user interface design, Videotex, management graphics. (961 Eglington Ave. E., Suite 200, Toronto, Ontario M4G 4B5; (416) 424-4758/3482)

The First Regional Conference on Technology and Special Education is being presented by the New York State Association for Educational Data Systems on May 22. Computer Applications in special education for administrators, teachers, and parents will be discussed at the Mill Neck Manor Lutheran School for the Deaf, Frost Mill Rd., Mill Neck, LI, NY 11765. Contact Dr. Dolores Shanahan of Commack Public Schools (Indian Hollow Computer Lab, Kings Park Rd., Commack, NY 11725; 516/493-3431) or Jerry Burke of Half Hollow Hills High School (Dix Hills, NY 11746; 516/549-6792).

A conference on **Local Area Networks** will be held May 25 and 26 at the Halloran House, New York City, by Frost & Sullivan, Inc., who describe themselves as "a leader in communications, data processing, and office of the future market research." (Future market research? Is that something like crystal-ball gazing?) Topics will include selecting the best LAN for your company, applications, LANs integrated into the office of the future, baseband vs. broadband, standards and protocols, voice integration and data tradeoffs, and prospects for PBX and CBX. Moderator will be D. R. MacNaughton, president of Business Development International, consultants in computers and data communications. MacNaughton is project director for Frost & Sullivan facsimile, electronic mail, and computer maintenance studies. They list no price for attending, but you should contact Account Representative Carol Sapchin. (106 Fulton St., New York, NY 10038; 212/233-1080)

The **Technical Education Research Centers** (TERC) are giving workshops on Microcomputers in Education at Gutman Library in Cambridge, Massachusetts, May 18-20 and at the Taft School in Watertown, Connecticut, June 7-9 for "professional development for educators at all levels, elementary through college." (Stuff like that is put in quotes because *I* don't know what "professional development" means and you shouldn't think that's my terminology.) Workshops will "emphasize hands-on experience," they say, on a "variety" of micros. Extensive workshop reference materials will be "given" to participants, and there will be evening symposia on current trends in microcomputer applications to education. Hotel accommodations will be "available." (8 Eliot St., Cambridge, MA 02138; 617/547-3890)

Virginia Polytechnical Institute and State University is offering a workshop on personal microcomputer interfacing and scientific instrument automation, May 31-June 4. Contact Dr. Linda Leffel. (Blacksburg, VA 24061; 617/547-3890)

The **MUMPS User's Group** are holding their eleventh annual conference June 7-11 at the Denver Hilton. (You'll remember them from last August's Of Interest. MUMPS is a microcomputer language used in medical and business environments.) Hardware, software, and systems vendors will be there. There will be presentations, tutorials, workshops, discussions. (Box 37247, Washington, DC 20013; 301/779-6555)

A workshop on the design and creation of "'user-friendly' computer programs for immediate use in classroom, office or home" will be conducted July 6-23 at Fresno State University by Dr. Dan Isaacson, Editor/Publisher of **CourseWare Magazine**. Price is $153, with additional unspecified charges for "materials" and other university fees. The course is aimed at teachers and business persons who want to "design innovative programs as effective teaching or working tools," ". . . develop effective computer-based learning experiences," and produce "useful and marketable" software. The magazine offers a scholarship refund of tuition and fees to each participant who completes the course and writes "a well-designed program accepted for publication" by the magazine. (*Caveat emptor.*) (Reader Service **No. 532**)

The National Conference on Artificial Intelligence will be August 16-20 at Carnegie-Mellon University and the University of Pittsburgh. Featured topics will be robotics and something called "expert systems." Growing out of AI research, expert systems are computer programs with large stores of specialized knowledge, to solve complex problems by imitating the decision-making processes of human experts and, in some cases, outperforming humans. Expert systems are now used in medicine, geology, and business, and will soon be available on home computers to help with tax forms, automobile maintenance and repair, etc. A two-day tutorial will give a nontechnical look at AI August 16 and 17. Other topics include computational vision, programmable automation, game playing, and knowledge representation (what?). Carnegie-Mellon's new Robotics Institute will be a focal point of the conference. Further information is available from the **American Association for Artificial Intelligence**. (Reader Service **No. 542**)

DDJ

Program for a Modem

Forth Code From the Conference Tree

by John S. James

John S. James, c/o CommuniTree Group, 470 Castro St., Suite 207-3002, San Francisco, CA 94114.

The modem is your computer's link to the telephone – a window to databases maintained by thousands of individuals and organizations. If you want to write your own program to control a modem, there are several different ways to proceed. You may use calls to the manufacturer's firmware supplied in the modem, you may ignore the firmware and program the modem hardware directly, or, if you have a "smart modem" (one with its own built-in microprocessor), you would communicate with the modem by using control sequences of ASCII characters.

This article concerns primarily the second alternative; programming a Hayes MicroModem II* for the Apple II** by directly accessing the modem chip, ignoring the firmware. This writeup may be useful to several groups of programmers:

(1) Those who want to program the MicroModem II directly, in any language, will find information here not readily available elsewhere.

(2) Those interested in programming a computer to control other external devices may find this example useful.

(3) The code shows various FORTH techniques in an application, not a teaching example.

This code is used in The Conference Tree, a program for computer conferencing. Anyone with an Apple II, 48K, one or more floppy disk drives, and a Hayes MicroModem II can sponsor computer conferences, which may be accessed by anyone with a 300 baud or 110 baud terminal (not necessarily an Apple). New messages are always attached to other messages already in the conference, forming a tree of messages structured by the users; any user's entry may become the start of a new conference. Various other features such as public and private conferencing, seven levels of password protection, and a "turing test," which allows the conference sponsor to make sure that only persons (not computers) log on, are provided. As far as is known, there is no way to crash the system or violate its security from a remote location. For more information on The Conference Tree, see "The Conference Tree" by this author, in *Dr. Dobb's Journal* #60, or write to The CommuniTree Group.

One caveat – the code released here is from a working system, but is not a polished teaching example. It could use substantial improvements. For reliability, we chose to release the code as is and explain how it should be improved, instead of giving it a facelift for publication.

The code published here has been placed in the public domain by The CommuniTree Group. It contains information not readily available elsewhere, thanks in large part to the excellent help we have received from Hayes Microcomputer Products in answering our questions and volunteering advice on matters we had overlooked. Remaining deficiencies are of course the sole responsibility of the author.

How the Program Is Organized

The Apple provides seven "slots" which the modem (or other peripheral device) may be plugged into. The slot numbers are 1 through 7. The modem has several one-byte control and status registers, which are accessed through addresses in the 6502 address space; the addresses are functions of the slot number.

The purpose of the modem control section of The Conference Tree program is to build up several operations which may be used later in the application code to do useful work. The most important of these operations are XKEY (read a character from the modem), XEMIT (write to the modem), GETCALL (do everything necessary to receive a phone call, matching the user's terminal baud rate and parity if any), and XHANGUP (hang up the phone).

A rudimentary knowledge of FORTH should be sufficient for reading most of this program. As we chose to implement The Conference Tree entirely in higher-level FORTH, there is no 6502 code.

Screen By Screen

FORTH source code is traditionally organized as "screens" of 1024 bytes (16 lines by 64 chracters). Since the Apple II uses 40-character lines, most Apple FORTH systems organize the screen as 24 40-column lines.

Screen 32

This "load screen" compiles the rest of the source code.

Screen 33

Two Apple-specific operations are defined. ?KEY tests whether a key has been pressed on the Apple, and returns a boolean, or truth value – "1" if a key has been pressed, "0" otherwise. On some (but not all) FORTH systems, ?TERMINAL does the same thing.

KEYLOOK returns the value of a key that has been typed, like the FORTH word KEY. But unlike KEY, it does not wait in a busy loop if no key has been pressed, but returns zero instead.

Because these definitions use address and mask values, it is best to define them in hexadecimal. Since we follow the convention that all references to screen numbers should be in decimal, we return to decimal before finishing the screen; otherwise, some numbers on the load screen would have to be in hex.

Screen 34

This screen defines PARITYTABLE, which has 16 entries, each zero or one, corresponding to whether the binary numbers 0000-1111 have even or odd parity. This table will be used for faster determination of the parity of an input or output byte.

Screen 35

RAMBYTES, defined in this screen, is an alternative to the FORTH word VARIABLE for allocating memory. It defines address constants, putting the data at any address desired and keeping it out of the FORTH dictionary, which should be kept ROMable. The author prefers RAMBYTES to VARIABLE for almost all uses, except for tables of constants which should be kept in the dictionary so that they will be loaded automatically (e.g., PARITYTABLE).

It is the author's personal convention to give flags names which end in "F" and to allocate two bytes for each, even though one would be enough, to avoid the danger of using 16-bit operators by mistake.

* MicroModem II is a registered trademark of Hayes Microcomputer Products, Inc.

** Apple II is a registered trademark of Apple Computer, Inc.

Screen 35 – The Variables

MODEMSLOT tells what slot number the modem is plugged into. It must be set before the program is run.

XSPEED keeps track of the program's toggling between 300 and 110 baud when a user is logging on.

TEMP will be used for the original input character. The value returned by the input word has the high-order bit stripped off. The original value is needed for checking parity when a user first logs on.

PARITY will indicate whether even or odd parity has been detected from the terminal.

CHARINDEX gives the current position in the 80-character input line. It is here only because the modem must not echo a character typed past end-of-line.

COUNT1 and COUNT2 are used for nested timeout loops, which automatically hang up the phone if the user has done nothing for about ten minutes.

CALL# counts the phone calls received since the system was last put online.

LOCALF indicates local-mode operation, i.e., using the system without the phone line as a local database.

ABORTF is set if the session must be aborted due to timeout.

NOLOGONF indicates that logons have been disabled, because the operator needs to take the system down at the end of the current session.

HALFDUPLEXF indicates halfduplex vs. fullduplex operation. Full (program does echo characters typed by the user) is assumed unless a command (TERMINAL) is used to set to half. If you want to handle both modes, it is better to start with fullduplex, as users will be less confused if the characters they type print twice than if they don't print at all.

Screen 36

Here the three registers used – the Ring, Status, and Data registers – are set up. These registers, on the ACIA chip used in the modem, are accessed through byte addresses in the 6502. Every reference to a register computes its address as a function of the slot number where the modem is plugged in. It would be more efficient to compute the address only once, but there is no problem keeping up with 30 characters per second in either case, so we chose to simplify the code instead of optimizing its speed.

For details on the registers, see the MicroModem II manual.

XINIT initializes the modem. It first puts the byte "3" into the status register, as required by the ACIA chip. Then it checks the XSPEED variable to decide how to initialize the status and ring registers – 300 or 110 baud. Later operations will reinitialize at each speed alternatively, while a new user is trying to get on, until a carriage return character is recognized. (The user is told to type two carriage returns to start using the system.)

Screen 37

X?HUNGUP tests whether the user has hung up (or the carrier has otherwise been lost).

First, the LOCALF flag is checked to see if the system is in local mode (operating without the modem or phone at all). If so, we cannot be "hung up," so a zero (boolean false) is returned. Otherwise we put a zero on the stack, then check the status register, doing a bitwise AND with binary 4 (00000100) to get the not-carrier-detect bit. If that bit is on, we reinitialize and test again in case it was only a momentary interruption (this may not be necessary); if it is still lost we drop the zero and replace it with a one (boolean true) to indicate that the user has hung up.

Screen 38

?ESCAPE is a special operation used to allow the operator of the conference to gracefully shut it down without interrupting a user online at the time. If a user is on when the operator needs to take down the system (e.g., to use the Apple for something else), the operator can press the ESC key, and the user will receive a message that the computer is needed in five minutes. Also, all further logons will be disabled. Alternatively, the operator may press Z, to silently disable further logons, allowing the current user to finish the session normally.

Parts of this operation are a bit complicated, and not necessary for understanding the modem software itself. The dummy operation is for allowing a forward reference to be patched in later; the operation which actually sends the "computer is needed" message to the user must be defined later in the program, after the modem operations are available. KEYLOOK returns the value from the buffer where the Apple key input is stored. If it is ESC or Z, ?TERMINAL DROP clears the keyboard strobe, 07 EMIT beeps the Apple speaker, and the NOLOGONF flag is set to boolean true, to disable any later logons. The operator is informed that logons are disabled. If the key was ESC, the user is warned, and a boolean true is returned by ?ESCAPE, indicating that the user's command has been interrupted and should be aborted.

Screen 39

Although the micromodem is able to set parity automatically, we chose to do all the work ourselves. EVENPARITY uses a table lookup of each four bits of the byte to set even parity quickly. SETPARITY tests the carriage returns initially typed by the user, and sets the PARITY variable to tell the program which parity to output. XKEYLOOK reads a key from the modem, except that if no key has been typed, it does not hang in a busy loop but returns "0", and if the phone has been hung up, it returns "256", an impossible value for any key to return, as an indicator.

Screen 40

XEMIT sends a character to the modem. XEMITP does the same, with even or odd parity applied. XHANGUP hangs up the phone. XWAITRING waits for the phone to ring; as explained below, this definition should be improved.

Screen 41

ECHO echos the character that the user typed back to the terminal, if the terminal is operating in fullduplex. Echo does not occur if the terminal is in halfduplex, or if a backspace was typed, or if the user was at end of line (column 80). Carriage returns also are not echoed, although the program should probably have been organized so that they would be, to make sure the user gets immediate feedback.

Screen 42

TIMEOUT? automatically hangs up the phone if the system has been waiting for input for about ten minutes. The main purpose is to protect users against forgetting to log off (hang up) when they have called long distance. It also prevents malicious users from disabling a tree by calling up and leaving the phone connected indefinitely. Incidentally, the timeout probably cannot be done through the

firmware supplied with the modem – one of the reasons we bypassed it and controlled the chip directly.

TIMEOUT? uses two nested loops to keep track of how long the system is in a busy loop waiting for a key to be pressed at the terminal (two loops were used since a single loop could not count high enough without using 32-bit numbers, which are not available in older implementations of FORTH). If timeout does occur, the phone is hung up, and the number on top of the stack is replaced with a non-zero value, which will terminate the busy loop in XKEY.

Screen 43

XKEY waits for a modem key to be typed. It must also check for timeout, for phone hung up (loss of carrier), and for the ESC or Z keys which may be pressed by the computer's operator. XKEY masks off the high bit of the character (7F AND), but saves a copy of the original byte in TEMP where it may be used to detect parity.

Screen 44

GETRING is part of GETCALL, screen 45. This section of the code should be cleaned up and perhaps reorganized.

GETRING prepares to wait for a call by making sure the phone is hung up and the modem is initialized. If logons have been disabled by the system operator (by using ESC or Z explained earlier), it informs the operator that the system is now clear, beeps (07 EMIT), and pauses (?KEY DROP KEY DROP); the ?KEY is to make sure no key struck earlier is waiting to be read. If the operator decides to allow further logons after all, any key may be pressed to continue.

On the operator's log (on the Apple Screen – it may also be directed to a printer) the program prints the call number, and the system status of waiting for a call. When the phone rings, the log shows that fact. In addition, the log shows the ASCII values of all characters received until a correct carriage return is recognized. (The XCR operation which prints this value is poorly named and placed in the code, but we chose not to change it now.)

Incidentally, for privacy reasons, the operator's log does not show what messages are read or written, only the commands used plus any erroneous commands or attempts to guess the password. The Conference Tree even allows private conferencing without the knowledge of the system operator, although the operator can set a password requirement to prevent it.

Screen 45

GETCALL makes sure the local-mode flag is off, i.e., the system is being used over the telephone. Then it goes into a loop to receive calls, and only terminates when a proper carriage return is detected. In other words, GETCALL will ignore any number of erroneous calls that may occur (e.g., voice calls, 1200 baud which the micromodem cannot use, etc.), and terminate when it receives a good call.

The loop does a GETRING, makes sure the keyboard strobe is clear, then toggles the speed between 300 and 110 baud, initializing each time. The "0" (not-timeout boolean) makes the UNTIL stay in the BEGIN . . . UNTIL loop unless a carriage return is found and the zero is replaced (DROP 1), terminating the loop. Parity is set from the carriage return character at this time.

Screen 46

SLOTTEST provides an error message if there is no card at all in the designated modem slot. The address of the slot's instruction space is computed, and read twice (DUP @ SWAP @). If there is no card, random values will be returned; if the values are different, an error message is given.

Testing

If you want to test-run this code, be sure to initialize MODEMSLOT, CHARINDEX, CALL#, LOCALF, ABORTF, NOLOGONF, and HALFDUPLEXF. Reasonable values are zero for everything except MODEMSLOT.

Execute GETCALL, then have someone with a terminal dial in and type a couple of carriage returns. You should be able to send ASCII characters, using XKEY and XEMITP.

Improvements

Since the program was written, we have learned of two recommended improvements in the modem control.

When the ring is detected, the program should wait until the ring finishes before picking up the phone, in order to avoid putting ring current over the modem contacts. Presumably the change can be made in XWAITRING. We have not had any problem with the present version, however.

Also, GETCALL should have a separate timeout, probably 30 seconds, to detect the carrier, and then timeout if no carriage return is received. The present timeout of approximately a minute leaves the system unavailable for that time in case of a voice call due to a wrong number. The screen may even show several lines of garbage as the phone company's message to please hang up comes on. No harm results, but future products should include this improvement.

Smart Modems

A smart modem is one which includes its own microprocessor. Control is usually accomplished by ASCII sequences communicated between the modem and the computer. The advantage of a smart modem is that it doesn't care what computer is used; it talks to a standard RS-232 interface.

CommuniTree Group is using a smart modem from Hayes Microcomputer Products for converting The Conference Tree to run under CP/M.

Anyone planning to use a smart modem should know that many serial cards use only three lines of the RS-232 cable – data in, data out, and ground. With these serial cards, the modem has no way to tell the computer that the user has hung up the phone, except by ASCII sequences which may be hard to distinguish from data typed by the user. Programs like The Conference Tree, which must run unattended, require that any serial or communications card used with a smart modem support at least the ability to detect carrier, and possibly other features such as ring detect. ▶▶J

(Listing begins next page)

```
SCR # 32

( LOAD SCREEN)

33 LOAD 34 LOAD 35 LOAD 36 LOAD
37 LOAD 38 LOAD 39 LOAD 40 LOAD
41 LOAD 42 LOAD 43 LOAD 44 LOAD
45 LOAD 46 LOAD
```

```
SCR # 33

( APPLE-SPECIFIC OPERATIONS)
HEX
: ?KEY ( ---BOOL. LIKE ?TERMINAL, BUT)
       ( TESTS ANY KEY)
   C000 C@ 80 AND
   IF 1   C010 C@ DROP   ELSE 0 THEN ;
: KEYLOOK ( ---CHAR-OR-0)
   ?KEY
   IF C000 C@ 7F AND ELSE 0 THEN ;
DECIMAL
```

```
SCR # 34

( MISC. DATA STRUCTURES)

( PARITY TABLE OF 16 0-OR-1 VALUES)
HEX
0 VARIABLE PARITYTABLE 0E ALLOT ( 16)
( THE FOLLOWING OPERATION 'T' IS A
  TEMPORARY COMPILE-TIME OPERATION
  WHICH IS DEFINED, USED TO LOAD THE
  TABLE CONVENIENTLY, AND THEN
  FORGOTTEN.)
: T ( 16# ---. MOVE NIBBLE PARITIES IN)
   10 0 DO PARITYTABLE I + C! LOOP ;
0 1 1 0  1 0 0 1  1 0 0 1  0 1 1 0  T
FORGET T
DECIMAL
```

```
SCR # 35

( VARIABLES)

: RAMBYTES ( ADDR N---ADDR. ALLOCATE)
   OVER CONSTANT + ;

24000 ( ADDRESS TO START THE DATA AREA)

2 RAMBYTES MODEMSLOT   ( SLOT #          )
2 RAMBYTES XSPEED      ( 300 OR 110      )
2 RAMBYTES TEMP        ( INPUT CHAR      )
2 RAMBYTES PARITY      ( ODD OR EVEN     )
2 RAMBYTES CHARINDEX   ( INPUT LN INDEX)
2 RAMBYTES COUNT1      ( FOR TIMEOUT     )
2 RAMBYTES COUNT2      ( FOR TIMEOUT     )
2 RAMBYTES CALL#       ( COUNT FOR LOG )
( FLAGS)
2 RAMBYTES LOCALF      ( TRUE=LOCAL OP )
2 RAMBYTES ABORTF      ( T=KILL COMMAND)
2 RAMBYTES NOLOGONF    ( T=LOGONS DISAB)
2 RAMBYTES HALFDUPLEXF ( T=HALF         )
( DATA AREAS)
81 RAMBYTES AREA$      ( INPUT BUFFER   )

CR  ." NEXT AVAILABLE ADDRESS:"   . CR
```

```
SCR # 36

( MODEM REGISTERS; INITIALIZATION)

HEX
: MMRING ( ---ADDR. MODEM RING & CNTRL)
   C085 MODEMSLOT @ 10 * + ;
: MMSTATUS ( ---ADDR. MODEM STATUS)
   MMRING 1+ ;
: MMDATA ( ---ADDR. MODEM DATA)
   MMRING 2+ ;
: XINIT ( ---. INITIALIZE)
   3 MMSTATUS C! ( MUST DO THIS FIRST )
   XSPEED @
   IF 8B 15 ( 300, NO PARITY ONE STOP)
   ELSE 8A 11 ( 110, TWO STOPS)
   THEN
   MMSTATUS C! MMRING C! ( SET SPEED) ;
DECIMAL
```

```
SCR # 37

( DETECT IF USER HANGS UP PHONE)

HEX
: X?HUNGUP ( ---BOOL. 1=HUNG UP)
   LOCALF @
```

(Continued on next page)

```
   IF 0 ( NOT 'HUNG UP' IF LOCAL)
   ELSE
     0    ( FLAG)
     MMSTATUS C@ 4 AND ( CARRIER LOSS?)
     IF XINIT MMSTATUS C@ 4 AND
         ( STILL NO CARRIER?)
       IF DROP 1 ( HUNG UP) THEN
     THEN
   THEN ;
DECIMAL
```

SCR # 38

```
( SESSION ESCAPE)

HEX
: DUMMY ;
: ESCMSG ( ---. DUMMY - PATCHED LATER)
   ( THIS WILL SAY COMPUTER IS NEEDED)
   DUMMY ;
: ?ESCAPE ( ---BOOL. T=ABORT THE CMND)
   KEYLOOK DUP 1B = OVER 5A = OR
   IF ( 'ESC' OR 'Z' KEY PRESSED)
     ?TERMINAL DROP    ( CLEAR STROBE)
     07 EMIT    ( BEEP TO OPERATOR)
     1 NOLOGONF !
     CR ." LOGONS DISABLED" CR
     1B =
     IF ( ESC KEY PRESSED)
       ." USER WARNED - 5 MINUTES" CR
       ESCMSG 1 ( ABORT)
     ELSE 0 ( NEEDN'T ABORT)
     THEN
   ELSE DROP 0 ( NEEDN'T ABORT)
   THEN ;
DECIMAL
```

SCR # 39

```
( MODEM PARITY; CHARACTER LOOK)

HEX
: EVENPARITY ( N---N. SET EVEN PARITY)
   DUP 000F AND PARITYTABLE + C@
   OVER 00F0 AND 10 / PARITYTABLE + C@
   +   1 =
   IF 80 + THEN ;
: SETPARITY ( ---. TEST FIRST C/R)
   TEMP C@ 80 AND 80 XOR PARITY ! ;
: XKEYLOOK ( ---N;0;256 XKEY NO WAIT)
   MMSTATUS C@ 1 AND
   IF MMDATA C@ 7F AND ELSE 0 THEN
   X?HUNGUP IF DROP 100 THEN ;
DECIMAL
```

SCR # 40

```
( MODEM WRITE; HANGUP PHONE; WAIT RING)

HEX
: XEMIT ( N---.)
   BEGIN MMSTATUS C@ 2 AND UNTIL
   MMDATA C! ;
: XEMITP ( N---. HANDLE PARITY)
   EVENPARITY   PARITY @ XOR ( ODD?)
   XEMIT ;
: XHANGUP ( ---. HANG UP PHONE)
   9 MMRING C! ;
: XWAITRING ( ---. WAIT FOR CALL)
   BEGIN   ( WAIT FOR RING)
     XHANGUP   MMRING C@ 80 AND 0=
   UNTIL   ( PHONE RINGS) ;
DECIMAL
```

SCR # 41

```
( ECHO, MISC)

: ECHO ( N---N.  ECHO TO REMOTE)
   ( DON'T ECHO IF: )
   HALFDUPLEXF @
   OVER 127 AND ( STRIP PARITY)
   DUP 8 = ( BACKSPACE)
   OVER 13 = OR ( CARRIAGE RETURN)
   SWAP 127 = OR ( ALSO BACKSPACE)
   OR ( HALFDUPLEX)
   CHARINDEX @ AREA$ 79 + = OR
                ( AT COL. 80)
   0= IF DUP XEMIT THEN ;
```

SCR # 42

```
( TIMEOUT IF USER DOES NOTHING)
```

```
HEX
: TIMEOUT? ( FLAG---FLAG.  TEST IF IDLE)
   1 COUNT1 +!   COUNT1 @ 80 =
   IF 0 COUNT1 !   1 COUNT2 +!
     COUNT2 @ 384 ( SECONDS - HEX) =
     IF  DROP 100
        XHANGUP   1 ABORTF !
     THEN
   THEN ;
DECIMAL
```

SCR # 43

```
( READ A CHARACTER FROM THE MODEM)

HEX
: XKEY ( ---N. 256=HUNG UP)
   0 COUNT1 !   0 COUNT2 !
   0 ( INITIAL VALUE)
   BEGIN
     MMSTATUS C@ 1 AND
     IF
       DROP MMDATA C@ ECHO
       DUP TEMP C! 7F AND
     ELSE
       X?HUNGUP IF DROP 100 THEN
       ?ESCAPE IF DROP 0D ( C/R) THEN
     THEN
     TIMEOUT?
   DUP UNTIL ;
DECIMAL
```

SCR # 44

```
( GETCALL - FIRST PART)

HEX
: GETRING ( ---)
   XHANGUP 3 MMSTATUS C! 15 MMSTATUS C!
   NOLOGONF @ ( NO FURTHER USERS ON?)
   IF CR ." SYSTEM IS CLEAR" 07 EMIT
     ?KEY DROP ( CLEAR STROBE)
     KEY DROP ( PAUSE FOR OPERATOR)
     0 NOLOGONF !
   THEN
   1 CALL# +!   ( INCREMENT CALL #)
   CR   ." #"   CALL# @ 3 .R
   ."   WAITING FOR CALL.."
   XWAITRING  ." RING" CR 6 SPACES ;
DECIMAL

: XCR ( ---. TYPE CHARACTER ON LOG)
MMSTATUS C@ 4 AND 0=
IF TEMP C@ . THEN ;
```

SCR # 45

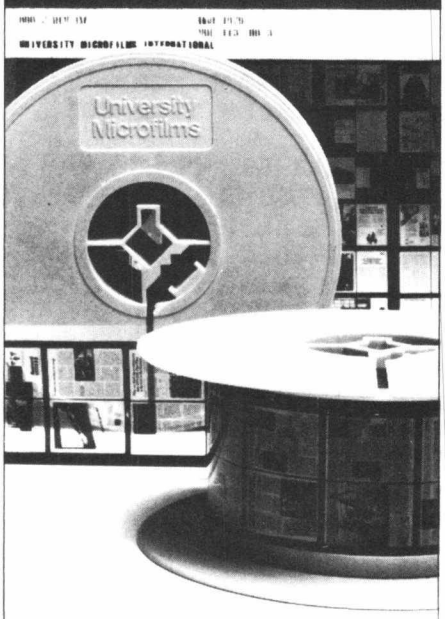
```
( GETCALL - RECEIVE A VALID CALL)

HEX
: GETCALL ( ---. RECEIVE PHONE CALL)
   0 LOCALF !  ( SET REMOTE MODE)
   BEGIN ( RECEIVING CALLS TILL 0D)
     GETRING
     ?KEY DROP ( CLEAR STROBE)
     0 XSPEED !   ( TOGGLE - 300 FIRST)
     ( TOGGLE SPEED TILL GET C/R )
     0 ( INIT NOT-TIMEOUT BOOL   )
     E00 0
     DO  ( 60 SECOND TIMEOUT)
       XSPEED 1 TOGGLE XINIT
       XKEY 0D =    XCR
       IF DROP 1   SETPARITY LEAVE THEN
     LOOP  ( TILL FIND CR OR TIME OUT)
   UNTIL ( NOT TIMED OUT)
   0D XEMITP ( RETURN THE C/R FOUND)
   ." CONNECTED"  ;
DECIMAL
```

SCR # 46

```
( TEST TO MAKE SURE SLOT NOT EMPTY)

HEX
: SLOTTEST ( ---.  ABORT IF NO CARD)
   C000 MODEMSLOT @ 100 * +
   DUP @ SWAP @ -
   IF   CR
     ." ERROR, NO CARD IN MODEM SLOT "
     MODEMSLOT ?
     CR ." RESTART PROGRAM"
     7 EMIT
     BEGIN AGAIN
   THEN ;
DECIMAL
```

End Listing

This publication is available in microform.



University Microfilms International

Please send additional information for _____

Name _____

Institution _____

Street _____

City _____

State _____ Zip _____

300 North Zeeb Road
Dept. P.R.
Ann Arbor, Mi. 48106
U.S.A.

30-32 Mortimer Street
Dept. P.R.
London W1N 7RA
England

A Simpler Way

North Star-to-Epson

A recent article (A. Bomberger, "Connect a Parallel Printer to Northstar," *DDJ* No. 62) describes the use of the Epson MX-80 with the North Star Horizon microcomputer. Several months ago I installed an Epson MX-80 F/T with Graftrax 80 on a North Star Horizon (II-Quad, 64K HRAM) and have been immensely satisfied with the results. I use the system with the latest release of North Star DOS (5.2) and with Lifeboat Associates' CP/M version 2.23a. One of the main functions of the system is to run Micropro's WordStar version 3.00. I have found this to be a very powerful and cost-effective system.

It should be pointed out that this configuration is more easily implemented than Bomberger indicates. I, like Bomberger, was at first concerned that the handshaking signals between the Horizon and the Epson were not directly compatible. I compared the data transmission signals of the Epson printer with those of a Centronics printer that had been successfully connected to the Horizon parallel output port without any hardware modification. As these data transmission signals were identical (the Epson printer is indicated to have a Centronics Standard Parallel Interface) I had hopes that the implementation of the Epson printer would be straightforward. In fact, the only hardware tailoring I found to be necessary was in the wiring of the parallel cable to connect the North Star with the Epson. I wrote my own drivers for the North Star DOS, for CP/M, and for WordStar; however, the drivers were practically identical and closely model those from an earlier release of the North Star DOS. The drivers permit the transfer of all 8 bits to the parallel port, and thus make full use of the graphics capability of the Epson printer. While I don't know how my drivers will hold up under an interrupt request (this is a concern of Bomberger), I do know that in the foreseeable future they will not be required to do so. The best comment I can make about this implementation is that it works. For other North Star Horizon/Epson MX-80 users, this system may be an easier and entirely satisfactory system to install.

by Larry Shoer

Larry Shoer, Department of Chemistry, Lake Forest College, Sheridan & College Roads, Lake Forest, IL 60045.

Table 1 is the pin-to-pin configuration of the parallel connector cable. The driver in Listing I is used for the North Star DOS. I also set the byte at location 0ABC to C9 (RET) and the byte at location 0ABD to 00 (NOP). This prevents the printer from being initialized with a carriage return when the system is booted. I prefer this arrangement; otherwise, the printer must be turned on for the system to complete its cold boot, whether or not the printer will actually be used.

To use graphics from North Star BASIC it is necessary to first initialize the printer by outputting the reset command:

```
PRINT #2, CHR$(27),CHR$(64) \
  REM RESET PRINTER (ESC @)
```

To prevent North Star BASIC from automatically inserting carriage returns in the middle of the graphics output, it is also a good idea to disable the automatic carriage return generation by outputting the following command:

```
LINE #2,80,0 \ REM DO NOT
  SEND AUTO CR TO PRINTER
  (DOS 5.2)
```

The driver in Listing II is used for CP/M, and requires the Epson ON LINE button to be cycled prior to printing a file. While some may find this an inconvenience, I prefer to leave this function. The requirement of cycling the ON LINE button forces me to check the paper status prior to the actual print operation. The driver in Listing III is used for WordStar. It overwrites the WordStar port driver location. It supports status checking and, therefore, allows efficient simultaneous printing and editing. The North Star port driver can be installed with the WordStar Patcher, which may be invoked under the WordStar INSTALL program. The labels used in the above code will be recognized by the WordStar Patcher. To complete this phase of the installation, the WordStar Patcher should be instructed to set CSWTCH: equal to 01 (this selects the newly installed driver) and to set HAVBSY: to FF (this indicates that a busy test is available). This driver also requires the Epson ON LINE button be to cycled prior to each printing operation. The requirement of cycling the ON LINE button forces me to check the paper status prior to the actual print operation. This is especially useful since different forms will be used for different correspondence.

DDJ

Connection

Table 1

North Star Parallel Output Connector			Epson MX-80 F/T	
signal name*	pin		pin	signal name
data bit 7	1	-	9	data 8
data bit 5	2	-	7	data 6
gnd	3	-	17	chassis gnd
data bit 2	4	-	4	data 3
data bit 0	5	-	2	data 1
PO-flag	6	-	-	----
ack	7	-	10	$\overline{\text{acknlg}}$
spare	8	-	1	$\overline{\text{strobe}}$
data bit 6	9	-	8	data 7
data bit 4	10	-	6	data 5
data bit 3	11	-	5	data 4
data bit 1	12	-	3	data 2
gnd	13	-	-	----
gnd	14	-	33	gnd
gnd	15	-	16	logic gnd

* Confifuration as preset by North Star Computers. No user modification is necessary. For a description of these signals see Horizon Computer System Manual (25011B) pages 42 to 44.

North Star/Epson

Listing I

```
0AC5                ;*PRINTER PARALLEL OUTPUT ROUTINE
0AC5                COUT2   EQU     $
0AC5    DB06                IN      6       ;READ MOTHERBOARD STATUS
0AC7    E601                ANI     1       ;MASK TO GET THE PO FLAG
0AC9    CAC50A              JZ      COUT2   ;PRINTER NOT YET READY
0ACC    78                  MOV     A,B     ;GET CHARACTER TO ACC
0ACD    D300                OUT     0       ;OUTPUT TO PRINTER
0ACF    3E20                MVI     A,20H   ;LOAD PO RESET COMMAND
0AD1    D306                OUT     6       ;RESET PO FLAG
0AD3    78                  MOV     A,B     ;CHAR EXPECTED IN ACC
0AD4    C9                  RET             ;
0AD5    00                  NOP             ;NOP USED TO MAINTAIN
0AD6    00                  NOP             ;ALIGNMENT WITH
0AD7    00                  NOP             ;OTHER DOS 5.2 ROUTINES
                                            ;REFERENCE:
                                            ;HORIZON COMPUTER SYSTEM
                                            ;MANUAL (25011B) PAGE 44
                                            ;SYSTEM SOFTWARE MANUAL
                                            ;(25013B) PAGE H-2
                                            ;SYSTEM SOFTWARE MANUAL
                                            ;ADDENDUM (25501F) P. 11
```

End Listing

(Listings II & III begin on next page)

North Star/Epson (Listing continued, text begins on page 26)

Listing II

```
                        ;This USER AREA is for a Horizon computer
                        ;as used in a 24K Lifeboat CP/M2 on North Star.

                        ;Change MSIZE to the memory size desired
0018 =                  MSIZE   EQU     24      ;Distribution system

                        ;COUT2:
                                ;Parallel port output.
5AE4    DB06                    IN      6       ;READ MOTHERBOARD STATUS
5AE6    E601                    ANI     1       ;MASK TO GET PO FLAG
5AE8    CAE45A                  JZ      COUT2   ;PRINTER NOT YET READY
5AEB    79                      MOV     A,C     ;GET CHAR TO ACC
5AEC    D300                    OUT     0       ;OUTPUT TO PRINTER
5AEE    3E20                    MVI     A,20H   ;LOAD PO RESET COMMAND
5AF0    D306                    OUT     6       ;RESET PO FLAG
5AF2    79                      MOV     A,C     ;CHAR EXPECTED IN ACC
5AF3    E67F                    ANI     7FH     ;MASK FOR ASCII CHAR
5AF5    C9                      RET             ;RETURN WITH CHAR IN ACC
5AF6    00                      NOP             ;NOP USED TO MAINTAIN
5AF7    00                      NOP             ;ALIGNMENT W/ HORUSER.ASM
5AF8    00                      NOP             ;(LIFEBOAT'S CP/M)
5AF9    00                      NOP             ;REFERENCE:
5AFA    00                      NOP             ;HOR. COMPUTER SYS MANUAL
5AFB    00                      NOP             ;(25011B) PAGE 44
5AFC    00                      NOP             ;SYSTEM SOFTWARE MANUAL
5AFD    00                      NOP             ;(25013B) PAGE H-2
5AFE    00                      NOP             ;SYSTEM SOFTWARE MANUAL
                                                ;ADDENDUM (25501F) P. 11

5B49    C9                      RET             ;CODE REMOVED TO OUTPUT
5B4A    00                      NOP             ;A CARRIAGE RETURN TO THE
5B4B    00                      NOP             ;PRINTER DURING COLD BOOT
5B4C    00                      NOP             ;NOP USED TO MAINTAIN
5B4D    00                      NOP             ;ALIGNMENT W/ HORUSER.ASM
                                                ;(LIFEBOAT'S CP/M)
```

End Listing

Listing III

```
0728    DB06    POBSY:  IN      6       ;INPUT MOTHERBOARD STATUS
072A    E601            ANI     1       ;STATUS 0 - NOT READY
                                        ;       1 - READY
072C    2F              CMA             ;REVERSE STATUS FLAG
072D    1F              RAR             ;PUT FLAG IN CARRY
072E    C9              RET             ;STATUS CHECK COMPLETE
072F    00              NOP             ;NOP'S MAINTAIN ALIGNMENT
0730    00              NOP             ;WITH WORDSTAR PORT
0731    00              NOP             ;DRIVER
0732    00              NOP             ;
0733    D300    POSEND: OUT     0       ;OUTPUT CHAR FROM A REG
0735    3E20            MVI     A,20H   ;LOAD PO RESET COMMAND
0737    D306            OUT     6       ;RESET PO FLAG
0739    C9              RET
073A    00              NOP             ;NOP'S MAINTAIN ALIGNMENT
....    ..              ...             ;WITH WORDSTAR PORT
....    ..              ...             ;DRIVER
0744    00              NOP
```

End Listing

Add a System Clock

For a variety of applications, a system clock in a computer system is desirable in order to keep track of the temporal sequence of events and their durations. Examples include records of computer usage and accounts, performance evaluation and time-slicing in a time-shared operating system. Frequently, several clocks are required to monitor various events simultaneously. For example, in performance analysis of a computer system, we would like to have a separate clock to record the system elapse time, CPU-utilization time, disk seek and latency times, job mean queue time or printer I/O time, etc.

Since the Cromemco computer does not provide any hardware clock, we have implemented a software simulated clock system in an independent, small Z80 or 8080 microcomputer, external to the host Cromemco computer.

Hardware

The clock simulator is implemented in a microcomputer consisting of four boards:

1. ZPU board
2. TUART board (serial I/O board)
3. BYTESAVER prom board
4. Memory board (8K RAM from Industrial Micro Systems).

The first three boards are from Cromemco since we have some extra Cromemco boards for backup purpose, but any other similar board can be used for the clock simulator instead. The BYTESAVER contains a Cromemco Z80 monitor (ZM-108) in a 1K ROM at Hex address E000. The ZPU board is operated at 2 MHz and is configured for power-on jump to Hex address E000, where the Z80 monitor resides. A CRT (9600 baud) is connected to the serial Port A of the TUART board as control during initialization. The clock simulator program itself resides in a 2708 PROM at Hex address F000 on the BYTESAVER board. To enter the clock simulator program from the Z80 monitor, one simply executes the "G" or GOTO command by entering

G F000 (Carriage return)

on the CRT.

by Michael Kan

Michael K. Kan, c/o Scripps Memorial Hospital, 9888 Genesee Ave., La Jolla, CA 92037.

At this point, the CRT can be disconnected and a 3-wire cable is used to connect serial Port A of the clock simulator to the clock serial port of the host computer. Pin 2 (send) and pin 3 (receive) of the clock simulators's serial port are connected to pin 3 (receive) and pin 2 (send) respectively of the host computer's serial port so that the clock simulator would act like a peripheral device to the host computer. Pin 7 of both serial ports are connected to signal ground.

The host computer used to test the clock simulator and for performance analysis is a Cromemco Z-2 computer equipped with 32K bytes RAM, a 4 FDC disk controller, a Persi 277 disk drive and a TUART board. One of the two serial ports on the TUART (Port B at Hex address 50) is used as a printer port. The other serial port (Port A at Hex address F0) is used as an I/O port for communication with the clock simulator. This port is operated at 9600 baud.

Program

The clock simulator program provides two independent 32-bit clocks to keep track of the system elapse time and printer I/O time used in performance analysis. More clocks can be easily added if required. Each clock is implemented in 4 bytes of RAM in the low memory while the clock simulator program (SOFTCLKR) resides in a 2708 PROM at Hex address F000 (Figure 1 on page 34). The assembly program of SOFTCLKR is listed in Figure 2 on page 34. The SOFTCLKR program consists of a timer loop (timer subroutine) which provides the necessary time delay for each clock increment. The exact timer delay depends on a counter value which can be varied to meet a wide range of needs. A counter value of 23 Hex or 35 decimal is used in the clock simulator. The clock interval in this case is measured to be 0.1844 millisecond.

The clock simulator is controlled by commands from the host computer sent over by way of the serial port and 3-wire cable. Command 0 resets both system and printer clocks, commands 1 and 2 start and stop the system clock while commands 3 and 4 start and stop the printer clock. Commands 5 and 6 retrieve the clock data from the system and printer clocks respectively.

The clock simulator scans the serial port for commands between the timing loops. The first five commands (0-4) are carried out by turning on or turning off the appropriate software switches stored in the lower RAM of the clock simulator. The software switches are represented by INITFG, SYSSW and PRTSW on the SOFTCLKR program listing. The clock simulator responds to commands 5 and 6 by sending 4 bytes of clock data (least-significant byte first) from the system and printer clocks respectively to the host computer. The maximum clock time depends on the clock interval. For a clock interval of 0.1 millisecond, the maximum clock time is 119 hours 18 minutes and 16 seconds.

Application

As an example for applying the clock simulator, we are going to go through the various steps in order to find out the system elapse time and printer I/O time in a prime number generator program. We shall assume that a simple BASIC program (PRIME1.BAS) as shown in Figure 3 on page 36 is used to generate the prime numbers.

Apart from the prime number generator program, two other programs are used to initialize the clock simulator and to print out the clock times at the end of the test program. These two programs and the test program (PRIME1.BAS) are stored in a commmand file for batch operation. The first program in the command file is a JOBBEG program implemented in FORTRAN (Figure 4 on page 37). This sends commands to the clock simulator to reset all clocks and turn on the system clock just before loading the test program specified in the command file. Each test program has incorporated instructions to turn on the printer clock just before a prime number is printed on the printer and turn off the printer clock after completion of printing.

The last program in the command file is a JOBEND program which sends a command to the clock simulator to stop the system clock and then gets the system and printer clock data and prints them out on the console. The listing of JOBEND program is given in Figure 5 on page 36. The host computer's console outputs are given in Figure 6 on page 38.

Discussion

Although our host computer is a Cromemco Z-2 computer, the clock

(Continued on page 57)

System Clock (Text begins on page 30)

Figure 1. 8080 machine codes for SOFTCLKR program. The program is stored in a 2708 PROM with starting address at F000.

```
F000: 31 E0 00 3E  00 32 F0 00  32 F1 00 3E  00 32 F2 00
F010: 32 F3 00 32  F4 00 32 F5  00 32 F6 00  32 F7 00 32
F020: F8 00 32 F9  00 32 FA 00  CD 97 F0 3A  FA 00 FE 01
F030: CA 0B F0 CD  78 F0 3A F0  00 FE 00 CA  57 F0 21 F5
F040: 00 34 C2 57  F0 2B 34 C2  57 F0 2B 34  C2 57 F0 2B
F050: 34 C2 57 F0  C3 87 F0 3A  F1 00 FE 00  CA 28 F0 21
F060: F9 00 34 C2  28 F0 2B 34  C2 28 F0 2B  34 C2 28 F0
F070: 2B 34 C2 28  F0 C3 87 F0  F5 D5 3E 00  11 23 00 1B
F080: BA C2 7F F0  D1 F1 C9 06  08 CD 78 F0  0B C2 88 F0
F090: 3E 07 D3 01  C3 87 F0 DB  00 E6 40 C8  DB 01 E6 7F
F0A0: FE 30 C2 AB  F0 3E 01 32  FA 00 C9 FE  31 C2 B6 F0
F0B0: 3E 01 32 F0  00 C9 FE 32  C2 C1 F0 3E  00 32 F0 00
F0C0: C9 FE 33 C2  CC F0 3E 01  32 F1 00 C9  FE 34 C2 D7
F0D0: F0 3E 00 32  F1 00 C9 FE  35 C2 0D F1  DB 00 E6 80
F0E0: CA DC F0 3A  F5 00 D3 01  DB 00 E6 80  CA E8 F0 3A
F0F0: F4 00 D3 01  DB 00 E6 80  CA F4 F0 3A  F3 00 D3 01
F100: DB 00 E6 80  CA 00 F1 3A  F2 00 D3 01  C9 FE 36 C2
F110: 42 F1 DB 00  E6 80 CA 12  F1 3A F9 00  D3 01 DB 00
F120: E6 80 CA 1E  F1 3A F8 00  D3 01 DB 00  E6 80 CA 2A
F130: F1 3A F7 00  D3 01 DB 00  E6 80 CA 36  F1 3A F6 00
F140: D3 01 C9 00  FF FF FF FF  FF FF FF FF  FF FF FF FF
F150: FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
```

Figure 2. SOFTCLKR program listed in 8080 assembly language.

```
; MK/10-23-80 SOFTCLKR.ASM (ROM BASED SOFTWARE SIMULATED CLOCK)
; CLOCK PROGRAM SIMULATED BY SOFTWARE WITH INCREMENT OF CLOCKS AT
; 0.1844 MILLISECOND USING THE TIMER SUBROUTINE.  THE CLOCK INTERVAL IS
; DETERMINED BY THE COUNTER VALUE STORED AT THE BOTTOM OF THE TIMER SUBROUTINE.
; IN THE CURRENT PROGRAM A COUNTER VALUE OF 23 HEX OR 35 DECIMAL IS USED AND
; THE CLOCK INTERVAL IS MEASURED TO BE 0.1844 MILLISECOND. OTHER CLOCK
; INTERVALS CAN BE OBTAINED BY ADJUSTING THE COUNTER VALUE, WHICH IS LOCATED
; NEAR THE END OF THE TIMER SUBROUTINE.
;
; TWO CLOCKS EACH CONSISTING OF 32 BITS OR 4 BYTES ARE AVAILABLE
; THE FIRST IS SYSTEM CLOCK (SYSCK) ENABLED BY THE SYSTEM CLOCK SWITCH (SYSSW)
; IF EACH CLOCK INCREMENT REPRESENTED 0.1 MILLISECOND, THE 32 BIT CLOCKS
; REPRESENT A MAXIMAL CLOCK INTERVAL OF 119 HOURS 18 MINUTES AND 16 SECONDS.
;
; THE SECOND CLOCK IS THE PRINTER CLOCK (PRTCK) ENABLED BY THE PRINTER
; CLOCK SWITCH (PRTSW). BOTH SYSTEM & PRINTER CLOCK SWITCHES ARE CONTROLLED BY
; SOFTWARE (OR ALTERNATIVELY BY AN INTERRUPT DRIVEN ROUTINE) WHICH CAN
; SELECTIVELY ENABLE OR DISABLE EITHER CLOCK SWITCHES BY SETTING ITS CONTENT
; TO 01H OR 00H RESPECTIVELY
; A THIRD SWITCH INITFG IS PRESENT AT LOCATION 0FAH.  WHEN IT IS SET TO 01H
; BOTH THE SYSTEM CLOCK AND THE PRINTER CLOCK WILL BE INITIALIZED.
;
; A LOCAL STACK IS USED BETWEEN 0E0H-0EFH
```

```
        DCX     H                       ;POINT AT NEXT HIGHER CLOCK BYTE
        INR     M                       ; UPDATE NEXT CLOCK BYTE
        JNZ     LOOP

        DCX     H
        INR     M
        JNZ     LOOP
        DCX     H                       ;POINT AT MOST SIGNIF. CLOCK BYTE
        INR     M
        JNZ     LOOP                    ;DONE
        JMP     OVFERROR                ;UNLESS THERE IS OVERFLOW ERROR
;
; SUBROUTINE FOR 0.1844 MILLISECOND DELAY, ASSUME 2 MEGA HERZ CLOCK RATE
;
TIMER   PUSH    PSW                     ;SAVE A REGISTER
        PUSH    D                       ;SAVE D REGISTERS
        MVI     A,ZERO                  ;CLEAR A REGISTER FOR COMPARE
        LXI     D,VALUE                 ;SET UP TIMER LOOP
TLOOP   DCX     D                       ;DECREMENT D&C 16 BIT PAIR
        CMP     D                       ;IS IT ZERO?
        JNZ     TLOOP                   ;IF NOT, LOOP BACK
DONE    POP     D                       ;RESTORE D
        POP     PSW                     ;RESTORE A
        RET
VALUE   EQU     23H                     ;VALUE=35 DECIMAL FOR 0.1844 MSEC DELAY
;


; OVERFLOW ERROR ROUTINE
; A BELL TONE WILL BE OUTPUTTED TO THE CONSOLE TO ALERT OVERFLOW CONDITION
;
OVFERR  MVI     B,008H                  ;COUNTER FOR TIME DELAY
        CALL    TIMER
        DCR     B                       ;TIME DELAY ENDS?
        JNZ     OVFERR+1                        ;NO, LOOP BACK
        MVI     A,07H                   ;GET ASCII VALUE FOR BELL
        OUT     01                      ;SEND TO CONSOLE (DEVICE NO=01)
        JMP     OVFERR                  ;LOOP FOR EVER
;
;
;
;
; SUBROUTINE TO READ INPUT PORT FOR COMMANDS FROM CP/M SYSTEM
; THE COMMANDS ARE USED TO START & STOP THE SYSTEM OR I/O CLOCKS
; INITIALIZATION OF THE CLOCKS OR ENQUIRY OF THE CLOCK DATA
; THE FOLLOWING SUBPROGRAM IS COMBINED TO THE SOFTCLK3 PROGRAM
; CLOCKCL1.ASM (CLOCK CONTROL PROGRAM)
;



; MK/10-15-80
; CLOCK CONTROL PROGRAM FOR UPDATING SWITCHES IN SOFTCLK3 PROGRAM
;
STATPT  EQU     000H            ;STAUS PORT ADDRESS
RDA     EQU     040H            ;RECEIVE DATA AVAILABLE MASK
TBE     EQU     080H            ;TRANSMIT BUFFER EMPTY MASK
DATAPT  EQU     001H            ;DATA PORT ADDRESS
RESETC  EQU     030H            ;ASCII 0, RESET ALL CLOCKS
BEGSCK  EQU     031H            ;ASCII 1, BEGIN SYSTEM CLOCK
STOSCK  EQU     032H            ;ASCII 2, STOP  SYSTEM CLOCK
BEGPCK  EQU     033H            ;ASCII 3, BEGIN PRINTER CLOCK
STOPCK  EQU     034H            ;ASCII 4, STOP  PRINTER CLOCK
```

```
; THE SYSTEM CLOCK IS LOCATED AT 0F2H-0F5H
; THE SYSTEM SWITCH IS LOCATED AT 0F0H
;
; THE PRINTER CLOCK IS LOCATED AT 0F6H-0F9H
; THE PRINTER SWITCH IS LOCATED AT 0F1H
; THE SOFTCLK PROGRAM STARTS AT LOCATION F000H
; THE SOFTCLK PROGRAM IS STORED IN A 2708 PROM FOR CONVENIENCE
;
STACK   EQU     0E0H
ZERO    EQU     000H            ;ZERO VALUE
ONE     EQU     001H            ;ONE  VALUE
INITF6  EQU     0FAH            ;0=NORMAL; 1=RESET CLOCK
SYSSW   EQU     0F0H            ;SYSTEM SWITCH 0=OFF, 1=ON
PRTSW   EQU     0F1H            ;PRINTER SWITCH 0=OFF, 1=ON
SYSCK   EQU     0F2H            ;SYSTEM CLOCK,4 BYTES 0F2H-0F5H, MS BYTE FIRST
PRTCK   EQU     0F6H            ;PRINTER CLOCK,4 BYTES, MOST SIGNIF. BYTE FIRST
;
        ORG     0F000H                  ;STARTING ADDRESS
BEGIN   LXI     SP,STACK                ;SET UP STACK
        MVI     A,ZERO                  ;CLEAR CLOCKS & SWITCHES
        STA     SYSSW                   ;DISABLE SYSTEM CLOCK SWITCH
        STA     PRTSW                   ;DISABLE PRINTER CLOCK SWITCH
INITCK  MVI     A,ZERO                  ;CLEAR SYSTEM CLOCK
        STA     SYSCK
        STA     SYSCK+1
        STA     SYSCK+2
        STA     SYSCK+3
        STA     PRTCK                   ;CLEAR PRINTER CLOCK
        STA     PRTCK+1
        STA     PRTCK+2
        STA     PRTCK+3
        STA     INITF6                  ;SET INIT CLOCK FLAG=0 (NON-INIT)
LOOP    CALL    RDPORT                  ;CHECK INPUT PORT FOR STATUS UPDATE
        LDA     INITF6                  ;CHECK INITIALIZATION CLOCK FLAG
        CPI     ONE                     ;IF FLAG IS SET(=1), INIT CLOCK
        JZ      INITCK
        CALL    TIMER                   ;IF NOT GET TIME DELAY
SYSTM   LDA     SYSSW                   ;CHECK SYSTEM CLOCK SWITCH
        CPI     ZERO                    ;IS SYSTEM SWITCH=0 OR OFF?
        JZ      PTRTM                   ;IF SO, SKIP SYSTEM CLOCK UPDATE
        LXI     H,SYSCK+3               ;ELSE POINT AT LEAST SIGNIF. CLOCK BYTE
        INR     M                       ;INCREMENT SYSTEM CLOCK
        JNZ     PTRTM                   ;DONE IF NO CARRY OVER TO NEXT BYTE
        DCX     H                       ;POINT AT NEXT HIGHER CLOCK BYTE
        INR     M
        JNZ     PTRTM
        DCX     H                       ;POINT AT NEXT HIGHER CLOCK BYTE
        INR     M
        JNZ     PTRTM
        DCX     H                       ;POINT AT MOST SIGNIF. CLOCK BYTE
        INR     M
        JNZ     PTRTM                   ;GO TO NEXT SECTION FOR PRINTER CLOCK
        JMP     OVFERR                  ;UNLESS SYSTEM CLOCK IS OVERFLOWED


; THE NEXT SECTION UPDATES THE PRINTER CLOCK IF THE PRINTER CLOCK
; FLAG IS ON OR =1
;
PTRTM   LDA     PRTSW                   ;CHECK PRINTER CLOCK SWITCH
        CPI     ZERO                    ;IS IT OFF
        JZ      LOOP                    ;IF SO, SKIP NEXT SECTION
        LXI     H,PRTCK+3               ;POINT AT LEAST SIGNIF. CLOCK BYTE
        INR     M                       ;UPDATE CLOCK
        JNZ     LOOP                    ;DONE IF NO CARRY OVER TO NEXT BYTE
```

(Continued on top of page 34)

```
GSTIME  EQU     035H            ;ASCII 5, GET SYSTEM CLOCK'S TIME
GPTIME  EQU     036H            ;ASCII 6, GET PRINTER CLOCK'S TIME
;


RDPORT  IN      STATPT          ;GET INPUT PORT STATUS
        ANI     RDA             ;IS DATA RECEIVED?



        RZ                      ;IF NOT, RETURN
        IN      DATAPT          ;YES, GET DATA
        ANI     07FH            ;MASK OFF PARITY BIT
        CPI     RESETC          ;RESET CLOCK COMMAND
        JNZ     BSYSCK          ;NO
        MVI     A,ONE           ;YES, SET INITIALIZATION FLAG=1(ON)
        STA     INITF6



        RET                     ;RETURN
BSYSCK  CPI     BEGSCK          ;START SYSTEM CLOCK COMMAND?
        JNZ     SSYSCK          ;NO
        MVI     A,ONE           ;YES, SET SYSTEM CLOCK SWITCH=1(ON)
        STA     SYSSW
        RET                     ;RETURN
SSYSCK  CPI     STOSCK          ;STOP SYSTEM CLOCK COMMAND?
        JNZ     BPRTCK          ;NO
        MVI     A,ZERO          ;YES, SET SYSTEM CLOCK SWITCH=0(OFF)
        STA     SYSSW
        RET                     ;RETURN
BPRTCK  CPI     BEGPCK          ;START PRINTER CLOCK COMMAND?
        JNZ     SPRTCK          ;NO
        MVI     A,ONE           ;SET PRINTER CLOCK SWITCH=1(ON)
        STA     PRTSW
        RET                     ;RETURN
SPRTCK  CPI     STOPCK          ;STOP PRINTER CLOCK COMMAND?
        JNZ     GETCLK          ;NO
        MVI     A,ZERO          ;SET PRINTER CLOCK SWITCH=0(OFF)
        STA     PRTSW



        RET                     ;RETURN
GETCLK  CPI     GSTIME          ;GET SYSTEM CLOCK COMMAND?
        JNZ     GETPTM          ;NO, CHECK GET PRINTER CLOCK COMMAND
GETSTM  IN      STATPT          ;YES, GET OUTPUT PORT STATUS
        ANI     TBE             ;IS TRANSMIT BUFFER EMPTY?
        JZ      GETSTM          ;NO,LOOP BACK
        LDA     SYSCK+3         ;YES, GET LEAST SIGNIF. CLOCK BYTE
        OUT     DATAPT          ;AND SEND IT OUT
GETST2  IN      STATPT
        ANI     TBE
        JZ      GETST2



        LDA     SYSCK+2         ;SEND 2ND LEAST SIGNIF. CLOCK BYTE
        OUT     DATAPT



GETST3  IN      STATPT
        ANI     TBE
```

(Continued on next page)

System Clock (Figures continued, text begins on page 30)

(Figure 2 continued)

```
        JZ      GETST3
        LDA     SYSCK+1         ;SEND 3RD LEAST SIGNIF. CLOCK BYTE
        OUT     DATAPT
GETST4  IN      STATPT
        ANI     TBE
        JZ      GETST4
        LDA     SYSCK
        OUT     DATAPT


        RET                     ;RETURN
GETPTM  CPI     GPTIME          ;GET PINTER TIME COMMAND(6)?
        JNZ     NEXTCM          ;IF NOT, SKIP PRINT CLOCK TIME PORTION
STAT1   IN      STATPT          ;IF TRUE, GET STATUS FOR OUTPUT
        ANI     TBE             ;IS TRANSMIT BUFFER EMPTY?
        JZ      STAT1           ;NO, LOOP BACK
        LDA     PRTCK+3 ;YES, GET LEAST SIGNIF. PRINTER CLOCK BYTE
        OUT     DATAPT
STAT2   IN      STATPT
        ANI     TBE
        JZ      STAT2
        LDA     PRTCK+2 ;SEND 2ND LEAST SIGNIF. PRINTER CLOCK BYTE
        OUT     DATAPT
STAT3   IN      STATPT


        ANI     TBE
        JZ      STAT3
        LDA     PRTCK+1         ;SEND 3RD LEAST SIGNIF. PRINTER CLOCK BYTE
        OUT     DATAPT
STAT4   IN      STATPT
        ANI     TBE
        JZ      STAT4
        LDA     PRTCK           ;SEND MOST SIGNIFICANT PRINTER CLOCK BYTE
        OUT     DATAPT
NEXTCM  RET                     ;RETURN (ADDITIONAL COMMANDS CAN BE ADDED HERE)
        END     BEGIN
```

Figure 3. First prime number generator (PRIME1) program in BASIC.

```
10 REM PRIME PROGRAM 1 IN CDOS 16K BASIC
20 REM MK/10-31-80
30 PRINT"          BASIC PRIME NUMBER GENERATOR 1"

40 INTEGER I,M,N,L
50 PRINT
60 INPUT"ENTER NUMBER OF PRIME NUMBERS DESIRED:  ",L
70 PRINT
80 PRINT"DIRECT PRIME NUMBER OUTPUT TO PRINTER"

 90 PRINT
100 OPEN\2,W\"$LP"
110 PRINT\2\"PRIME NUMBERS:"
120 PRINT\2\
130 PRINT\2\"               1"
140 PRINT\2\"               2"
150 REM L=NO. OF PRIME NUMBERS TO BE GENERATED
```

```
C
C INITIALIZED I/O PORT FOR CLOCK SIMULATOR
C
        WRITE(5,20)
20      FORMAT(1X,'INITIALIZE I/O PORT FOR CLOCK SIMULATOR',/)
C
C RESET CLOCK PORT AT HEX ADDRESS OF F2
C
        OUTPT=PORTA+2
        CALL OUT(OUTPT,RESET)
C
C DISABLE INTERRUPTS FROM CLOCK PORT
C
        OUTPT=PORTA+3
        CALL OUT(OUTPT,IMASK)
C
C SET BAUD RATE AT 9600
C
        OUTPT=PORTA
        CALL OUT(OUTPT,BAUD)
C
C RESET SYSTEM & PRINTER CLOCKS BY SENDING A CLEAR CLOCK COMMAND (0)
C TO THE CLOCK SIMULATOR
C
        WRITE(5,30)
30      FORMAT(1X,'RESET SYSTEM & PRINTER CLOCKS',/)
        OUTPT=PORTA+1
        CALL OUT(OUTPT,CLEARC)
C
C START SYSTEM CLOCK BY SENDING COMMAND 1 TO CLOCK SIMULATOR
C
        WRITE(5,40)
40      FORMAT(1X,'START SYSTEM CLOCK',/)
        CALL OUT(OUTPT,BEGSCK)
        END
```

Figure 5. Job end (JOBEND) program written in FORTRAN.

```
C MK/11-1-80 JOBEND.FOR
C FORTRAN JOB END PROGRAM TO STOP THE SYSTEM CLOCK OF THE CLOCK SIMULATOR
C AND TO PRINT OUT THE SYSTEM CLOCK TIME AND PRINTER CLOCK TIME BY
C CALLING A SUBROUTINE RDCLK (CLOCKR2.FOR).  THE TIME INTERVAL FOR EACH
C CLOCK INCREMENT IS 0.1844 MILLISECOND. THIS IS OBTAINED BY CALIBRATION
C OF THE CLOCK SIMULATOR USING A STOP WATCH.  THE CLOCK INCREMENT CAN BE
C ADJUSTED BY CHANGING THE CLOCK COUNTER IN THE TIMER SUBROUTINE OF THE
C SIMULATED CLOCK
C
        PROGRAM JOBEND
C
        INTEGER HOUR,MINUTE,SECOND
        REAL TIMINC
        BYTE PORTA,OUTPT,ENDSCK
        BYTE FIVE,SIX,TWO
        DATA FIVE,SIX,TWO/Z'35',Z'36',Z'32'/
        DATA PORTA,ENDSCK/Z'F0',Z'32'/
        DATA TIMINC/0.0001844/
C PORTA = BASE ADDRESS OF CLOCK SIMULATOR
C FIVE  = COMMAND TO GET 4 BYTES DATA FROM SYSTEM CLOCK
C SIX   = COMMAND TO GET 4 BYTES DATA FROM PRINTER CLOCK
```

```
160 REM N=CURRENT NUMBER TO BE TESTED AS PRIME NUMBER
170 REM M=DIVISOR USED TO CHECK N TO BE PRIME NUMBER OR NOT
180 REM IF M IS NOT DIVISIBLE BY 2 THROUGH N-1, IT IS A PRIME NUMBER
190 REM I=NOL OF PRIME NUMBERS GENERATED SO FAR
200 I=2

210  FOR N=3 TO 32767.0
220   FOR M=2 TO(N-1)
230   IF N=(N/M)*M THEN GOTO 330
240   NEXT M
250  REM N IS NOT DIVISIBLE BY ANY NO. FROM 2 TO N-1, IT IS A PRIME NO.
260  REM TURN PRINTER CLOCK ON BY SENDING COMMAND 3 TO CLOCK PORT
270  OUT%00F1%,%0033%
280  PRINT\2\USING"###############",N
290  REM TURN PRINTER OFF BY SENDING OUT COMMAND 4 TO CLOCK PORT

300  OUT%00F1%,%0034%
310  I=I+1
320  IF I=L THEN GOTO 340
330  NEXT N
340 PRINT"NO. OF PRIME NUMBERS GENERATED  ",I
345 PRINT
346 PRINT
350 CLOSE\2\
360 BYE
```

Figure 4. Job begin (JOBBEG) program written in FORTRAN.

```
C MK/11-1-80 JOBBEG.FOR
C JOB BEGIN FORTRAN PROGRAM TO INITIALIZE THE TUART SERIAL PORT FOR
C A BAUD RATE OF 9600.  THIS PORT IS USED FOR THE CLOCK SIMULATOR.
C THE PROGRAM WILL ALSO OUTPUT A CLEAR CLOCK COMMAND (HEX 30 OR ASCII 0)
C TO THE CLOCK SIMULATOR TO RESET BOTH SYSTEM CLOCKS AND PRINTER CLOCKS
C FINALLY THE PROGRAM STARTS THE SYSTEM CLOCK BY SENDING A START SYSTEM


C CLOCK COMMAND (HEX 31 OR ASCII 1) TO THE CLOCK SIMULATOR.
C
      PROGRAM JOBBEG.FOR
C
       BYTE BAUD
       BYTE OUTPT,PORTA,RESET,IMASK,BEGSCK
       DATA PORTA,RESET,IMASK,BEGSCK/Z'F0',Z'01',Z'00',Z'31'/
       DATA CLEARC,BAUD/Z'30',Z'C0'/


C
C BAUD  = BAUD RATE FOR 9600 BAUDS (HEX C0 TO BE SENT TO BAUD RATE RESET PORT)
C OUTPT = OUTPUT PORT (F1 FOR DATA,F2 FOR RESET, F3 FOR INTERRUPT RESET)



C PORTA = BASE ADDRESS OF TUART SERIAL PORT A SET AT HEX ADDRESS OF F0.
C RESET = HEX 01 TO RESET TUART
C IMASK = HEX 00 TO DISABLE INTERRUPTS
C BEGSCK= HEX 31 OR ASCII 1, COMMAND TO START SYSTEM CLOCK
C CLEARC= HEX 30 OR ASCII 0, COMMAND TO CLEAR SYSTEM & PRINTER CLOCKS
```

(Continued on top of page 36)

```
C FIVE AND SIX ARE REPRESENT BY HEX 35 (ASCII 5) AND HEX 36 (ASCII 6)
C
C


       WRITE(5,10)
10     FORMAT(1X,'STOP SYSTEM CLOCK',/)
       OUTPT=PORTA+1
       CALL OUT(OUTPT,TWO)


C
C READ SYSTEM CLOCK
C


       CALL OUT (OUTPT,FIVE)
       CALL RDCLK(HOUR,MINUTE,SECOND,TIMINC)
       WRITE(5,175) HOUR,MINUTE,SECOND
175    FORMAT(1X,'SYSTEM CLOCK TIME=  ',I4,' HOURS ',I3,' MINUTES ',
     /         I3,' SECONDS',/)


C
C READ PRINTER CLOCK
C


       CALL OUT (OUTPT,SIX)
       CALL RDCLK(HOUR,MINUTE,SECOND,TIMINC)
       WRITE(5,185) HOUR,MINUTE,SECOND
185    FORMAT(1X,'PRINTER CLOCK TIME= ',I4,' HOURS ',I3,' MINUTES ',
     /         I3,' SECONDS',/)
       END


C MK/10-28-80 CLOCKR2.FOR
C CLOCK SUBROUTINE 2 FOR READING 4 BYTES OF CLOCK DATA FROM THE
C CLOCK SIMULATOR THROUGH THE TUART SERIAL PORT A (ADDRESS AT HEX F0)
C THIS ROUTINE RETURNS THE TIME IN HOURS, MINUTES & SECONDS TO THE
C CALLING PROGRAM, WHICH PROVIDES THE ARGUMENT TIMINC OR THE TIME IN
C SECONDS FOR THE DURATION OF EACH CLOCK INCREMENT
C


C
       SUBROUTINE RDCLK(HOUR,MINUTE,SECOND,TIMINC)
C
       INTEGER MINUTE,SECOND,HOUR
       REAL TIMINC,SECBY1,SECBY2,SECBY3,SECTOT,MINTOT,HRTOT
       INTEGER BYTE1,BYTE2,BYTE3
       BYTE  CLKBY1,CLKBY2,CLKBY3,CLKBY4
       BYTE STATPT,DATAPT,INDATA,RAVAIL,STATUS
       DATA STATPT,DATAPT,RAVAIL/Z'F0',Z'F1',Z'40'/


C
C STATPT IS THE STATUS PORT FOR THE CLOCK SIMULATOR AT HEX ADDRESS F0
C DATAPT IS THE DATA PORT FOR THE CLOCK SIMULATOR AT HEX ADDRESS F1
C RAVAIL IS THE RECEIVE DATA AVAILABLE MASK
C INDATA IS THE DATA TO BE READ FROM THE STATUS PORT


C
C THE 32 BIT CLOCK IS REPRESENTED BY 4 BYTES FROM LEFT TO RIGHT
C CLKBY1,CLKBY2,CLKBY3,CLKBY4.
C
C READ 4 BYTES OF CLOCK DATA FROM THE CLOCK SIMULATOR.  THE LEAST
```

(Continued on next page)

System Clock (Figures continued, text begins on page 30)

(Figure 5 continued)

```
C SIGNIFICANT BYTE(CLKBY4) IS FIRST SENT BY THE CLOCK SIMULATOR FOLLOWED
C BY THE MORE SIGNIFICANT BYTES (CLKBY3,CLKBY2,CLKBY1)
C
C FIRST CHECK STATUS OF STATUS PORT(F0) UNTIL A CLOCK DATA BYTE ARRIVES
C THEN READ THE DATA INTO CLKBY4, CLKBY3, CLKBY2 & CLKBY1 RESPECTIVELY


C
200     INDATA=INP(STATPT)
        STATUS=(INDATA.AND.RAVAIL)
        IF(STATUS.EQ.0) GOTO 200
C
C DATA IS READY, GET DATA
C
        CLKBY4=INP(DATAPT)



C
C WAIT FOR NEXT BYTE
C
210     INDATA=INP(STATPT)
        STATUS=(INDATA.AND.RAVAIL)
        IF (STATUS.EQ.0) GOTO 210
        CLKBY3=INP(DATAPT)


C
C GET NEXT CLOCK BYTE
C
220     INDATA=INP(STATPT)
        STATUS=(INDATA.AND.RAVAIL)
        IF (STATUS.EQ.0) GOTO 220
        CLKBY2=INP(DATAPT)


C
C GET MOST SIGNIFICANT BYTE
C
230     INDATA=INP(STATPT)
        STATUS=(INDATA.AND.RAVAIL)
        IF (STATUS.EQ.0) GOTO 230
        CLKBY1=INP(DATAPT)
C
C AFTER READING THE 4 BYTES OF CLOCK DATA FROM THE CLOCK SIMULATOR,
C NOW CONVERT THE 32 BIT CLOCK DATA TO HOURS, MINUTES & SECONDS

C
C THE EQUIVALENT NO. OF SECONDS IN THE FIRST THREE MOST SIGNIFICANT
C BYTES ARE REPRESENTED BY SECBY1,SECBY2,SECBY3
C SINCE THE TIME INCREMENT IS VERY SHORT(0.1-1 MILLISECOND), THE LEAST
C SIGNIFICANT CLOCK BYTE (CLKBY4) IS NOT PROCESSED SINCE ITS MAX VALUE
C IS MUCH LESS THAN 1 SECOND
C


C CDOS FORTRAN REPRESENTATION OF A BYTE IS FROM -128 TO 127 SO THAT
C OFFH WILL REPRESENT -1 RATHER THAN 256.  TO CONVERT FROM -128 TO 127
C REPRESENT TO 0-256 REPRESENTATION, FIRST CHECK IF THE BYTE IS NEGATIVE
C IF SO, JUST ADD 256 TO THE NUMBER
C


        IF (CLKBY3.LT.0) BYTE3=CLKBY3+256
        IF (CLKBY2.LT.0) BYTE2=CLKBY2+256
        IF (CLKBY1.LT.0) BYTE1=CLKBY1+256
        IF (CLKBY3.GE.0) BYTE3=CLKBY3
        IF (CLKBY2.GE.0) BYTE2=CLKBY2
        IF (CLKBY1.GE.0) BYTE1=CLKBY1
C
        SECBY3=FLOAT(BYTE3)*(256.0*TIMINC)
        SECBY2=FLOAT(BYTE2)*(256.0*TIMINC*256.0)
        SECBY1=FLOAT(BYTE1)*(256.0*TIMINC*256.0*256.0)
C
C TOTAL TIME IN SECOND (SECTOT)
C
        SECTOT=SECBY1+SECBY2+SECBY3
C
C CALCULATE TOTAL TIME IN MINUTES (MINTOT)
C
        MINTOT=SECTOT/60.0
C
C CALCULATE TOTAL TIME IN HOURS (HRTOT)
C
        HRTOT=MINTOT/60.0


C
C CALCULATE SECOND,MINUTE & HOUR AS BELOW
C
        MINUTE=IFIX(MINTOT)
        SECOND=IFIX(SECTOT-FLOAT(MINUTE)*60.0)
        HOUR=IFIX(HRTOT)
        MINUTE=IFIX(MINTOT-FLOAT(HOUR)*60.0)
        RETURN
        END
```

Figure 6. Console output for BASIC PRIME1 program.

```
A,@ PRIME1BS
@ (BATCH) VERSION 01.03

A,JOBBEG

INITIALIZE I/O PORT FOR CLOCK SIMULATOR

RESET SYSTEM & PRINTER CLOCKS

START SYSTEM CLOCK

A,BASIC PRIME1.BAS

CDOS 16K BASIC, VERSION 5.4
        BASIC PRIME NUMBER GENERATOR 1
ENTER NUMBER OF PRIME NUMBERS DESIRED:  500

DIRECT PRIME NUMBER OUTPUT TO PRINTER

NO. OF PRIME NUMBERS GENERATED          500


A,JOBEND

STOP SYSTEM CLOCK

SYSTEM CLOCK TIME=      3 HOURS   1 MINUTES  40 SECONDS

PRINTER CLOCK TIME=     0 HOURS   4 MINUTES  42 SECONDS
```

End Listing

New USER for CP/M

Have you wanted to put computer listings on single sheets of paper to file in a note book or copy on a copying machine? The PIP utility in CP/M does this very well on continuous form, fan-folded paper but forget it with singe sheets! NEW USER will, among other things, allow you to print output on single sheets of paper that were previously designed for form-feed controlled, tractor printers.

The source listing, NEW USER, is a standard CP/M USER AREA assembled to operate on my computer. Like any USER AREA listing, it is unique to my computer. You will likely *not* be able to directly copy it unless you use an IMSAI MIO for device I/O, an IMSAI VIO-C for video display and an MX-80 F/T as the listing device.

However, the main feature of trapping form feeds should be easily implemented on any other computer system in the CP/M environment. The warm- and cold-boot initialization section will only be helpful if you use the IMSAI VIO-C and haven't yet figured how to keep the VIO-C out of "upper-case-only" mode or haven't fixed the reverse video problem found in early releases of this otherwise fine video board.

Simply speaking (Doc told me to keep it short and simple), the console input routine checks each input character for a FORMKEY character. If a FORMKEY character is found coming from the keyboard then the FORMBYT is toggled. That is, if it was an ASCII dollar-sign it becomes an ASCII space and vice-versa. This "switch" at FORMBYT selects and de-selects the form-feed trapping feature as well as directing the WARM BOOT message to either display a single line (no form feed trapping) or a second line (trapping form feeds).

In the printer (or listing) output routine, each character sent to the listing device is checked to see if it is a form feed. If it is *not* a form feed, control passes over the trapping routine and directly to the output routine. If the output character *is* a form feed, then a second check is made to see if the form feed trap "switch" is ON. If the byte at location FORMBYT is an ASCII space, then the trap is indeed ACTIVE. If the byte at location FORMBYT is an ASCII dollar sign, then the trap is not active and control again passes over the trapping routine.

In the trapping routine, printing output is suspended and a message is sent to the console notifying the operator that a form feed has been trapped and instructing the operator to press the carriage-return key to continue printing. This is the time to load another single sheet of paper before pressing the carriage return key.

Note: Select a FORMKEY value that doesn't interfere with any of CP/M's basic operations. I chose control-F. The integrity of the USER area is maintained except form feeds are cancelled and never reach the list device.

I have used NEW USER for several months with solid results. Dot PRN files generated by MAC with embedded form feeds can be easily sent to the printer by activiating the form feed trap and using the TYPE command with a control-P to turn on the printer. PIPing files with the [P] option works well with the form feed trap option.

If you run into any hidden problem with this that I overlooked, please let me and the readers of *DDJ* hear from you.

DDJ

(Listing begins on page 43)

by Gene Head

New USER for CP/M (Text begins on page 42)

```
;Software from --> Head Quarters - Corvallis, Oregon  97330

;NEWUSER.ASM  for CP/M2 on North Star

;      UP-DATES AND REVISIONS

;Version 3.4 of March 23, 1981
;Version 3.5 of August 22, 1981
;MX-80 F/T DRIVER INSTALLED 8-22-81
;FIX VIO-C HI-LITE PROBLEM  12-22-81

;OPTIONALLY TRAP FORM FEEDS 1-1-82
;Trapping F/F's is dynamically set with the FORMKEY key.
;Warm boot message will reflect if F/F's are trapped

;This version is for CP/M 2.2xx

;Change MSIZE to the desired CP/M memory size in K.

MSIZE   EQU     56              ;Put your size here
;
;Procedure is as follows:
;       0. Assemble NEWUSER.ASM but don't LOAD it
;       1. Run MOVCPM XX to the desired size (XX)
;       2. SAVE 40 CPMXX.COM
;       3. DDT CPMXX.COM
;               A. INEWUSER.HEX
;               B  R(OFFSET)      --> --> --> --> --> --> -->*
;               C. Control-C exit from DDT                    *
;       4. SYSGEN (DO NOT GET SYSTEM ONLY PUT SYSTEM)         *
;       5. SAVE 40 CPMXX.COM                                  *
;                                                             *
;The following equates are automatically changed by MSIZE.    *
                                        ;                     *
BIOS    EQU     5300H+(MSIZE-24)*1024   ;                     *
BDOS    EQU     05H                     ;                     *
USER    EQU     BIOS+700H       ;For double/quad density      *
;USER   EQU     BIOS+500H       ;For single density           *
OFFSET  EQU     2000H-BIOS      ;To overlay SYSGEN IMAGE <---*

FALSE   EQU     0
TRUE    EQU     NOT FALSE

FORMKEY EQU     'F'-40H         ;Key used to flip the trap
FORMTRAP        EQU     TRUE    ;Conditionally assemble TRAP

IOBYT   EQU     3               ;Store IOBYT
VIDEO   EQU     0F803H          ;VIO-C entry point
TTYDATA EQU     02              ;Printer is the MX-80 F/T
TTYMASK EQU     10H             ;Printer NOT BUSY if HIGH
STATUS  EQU     03              ;Status port on MIO card
KEYDATA EQU     02              ;Parallel data port MIO card
KEYMASK EQU     02              ;Parallel port MASK MIO card
VIOFLAG EQU     0F794H          ;U/L case byte on VIO-C card
```

(Continued on next page)

```
VIOBYTE EQU     28H             ;Sets VIO-C to U/L case mode
CLEAR   EQU     0F800H          ;VIO-C clear and home cursor
ASCMASK EQU     07FH            ;Mask out bit 7
BELL    EQU     07              ;ASCII Bell
NULL    EQU     00              ;ASCII NULL
CR      EQU     0DH             ;ASCII C/R
FF      EQU     0CH             ;ASCII F/F
ESCAPE  EQU     1BH             ;ASCII Escape key
HIBYTE  EQU     0F782H          ;VIO-C hi-lite byte
HIMASK  EQU     80H             ;Hi-lite bit mask


;       MODE BYTE OPTIONS

AUTOW   EQU     02 AND FALSE    ;Run AUTO on Warm Boot?
AUTOC   EQU     01 AND FALSE    ;Run AUTO on Cold Boot?
SINGLE  EQU     80H AND FALSE   ;Single Drive system?
WRITECK EQU     40H AND FALSE   ;Read after Write check?
ENINT   EQU     10H AND FALSE   ;Enable interupts

        ORG     USER-1
MODE    DB      AUTOW OR AUTOC OR SINGLE OR WRITECK OR ENINT
;
;80H sets printer to LPT: device, 0 sets to TTY:.
;STAT.COM can modify IOBYT in running CP/M.

DEFIOB  EQU     0               ;"I" Default IOBYT

NULLS   EQU     0               ;"J" Printer nulls
HAND    EQU     0

        ORG     USER


; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;JUMP TABLE - Jumps MUST remain here in same order.       *
CINIT   JMP     CINITR          ;Cold boot init          *
                                                         *
WINIT   JMP     WINITR          ;Warm boot init          *
                                                         *
CONST   JMP     UCONST          ;Console status          *
                                                         *
CONIN   JMP     UCONIN          ;Console input           *
                                                         *
CONOUT  JMP     UCONOUT         ;Console output          *
                                                         *
LIST    JMP     ULIST           ;Printer output          *
                                                         *
PUNCH   JMP     UPUNCH          ;Punch output to console*
                                                         *
READER  JMP     UCONIN          ;Reader input to console*
                                                         *
LISTST  JMP     ULISTST         ;Printer status          *

;
```

```
                                                                    *
;This 8 byte data area used externally MUST remain.                 *
LENUA:  DW      USRLEN          ;Length of USER AREA                *
USRIOB: DB      DEFIOB          ;"I" Initial IOBYT                  *
HSTYPE: DB      HAND            ;"H" Handshaking type               *
NULLOC: DB      NULLS           ;"J" Printer nulls                  *
        DB      0,0,0           ;Reserved                           *
;                                                                   *
; * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
;  This is the console status routine
;  When called by CP/M it will return with
;  Zero in the Accumulator if no data is waiting
;  and FF in 'A' if data is waiting.



UCONST:
        IN      STATUS          ;Get keyboard status
        ANI     KEYMASK         ;Mask data waiting bit
        MVI     A,0             ;00 in 'A' means no data
        RZ                      ;Return from here if NO DATA
        CMA                     ;'A' = OFF IF data waiting
        RET                     ;Return


;  This routine will return a character from the
;  keyboard in the ACC

UCONIN:
        IN      STATUS          ;Check keyboard status
        ANI     KEYMASK         ;Mask parallel port bit
        JZ      UCONIN          ;Wait until character typed
        IN      KEYDATA         ;Get keyboard data
        ANI     ASCMASK         ;Mask off bit 7

        CPI     FORMKEY         ;Is this a F/F flip key?
        JNZ     CONDONE         ;SINGLE EXIT POINT

;Process a control-F to filp the forms-trap function
        LDA     FORMBYT
        XRI     4               ;'$' --> ' '  OR  ' ' --> '$'
        STA     FORMBYT
        MVI     A,FORMKEY       ;Restore FORMKEY character

CONDONE RET                     ;Exit with character in ACC

;  This is the console or Video output routine

UCONOUT:
        PUSH    B               ;Save B,C
        MOV     A,C
        ANI     HIMASK          ;Get reverse video bit
        MOV     B,A             ;Save reverse video bit
        LDA     HIBYTE
        ANI     7FH             ;Get reverse video control byte
        ORA     B               ;Set or Reset Bit in control byte
        STA     HIBYTE
```

(Continued on next page)

```
        MOV     A,C
        CPI     BELL
        JNZ     UOUT            ;Force all bells to printer even
        OUT     TTYDATA         ;if printer is off line
UOUT    CALL    VIDEO           ;Put character on screen
        POP     B               ;Restore B,C
        MOV     A,C             ;RESTORE OUTPUT FOR CP/M 1.4
        RET
;
        ;  MX-80 F/T  routine

ULIST:
        MVI     A,FF
        CMP     C               ;Test for FORM FEEDS
        JNZ     LISTOUT         ;Ignore all except form feeds

        LDA     FORMBYT


        CPI     '$'             ;IF '$' THEN FORM
        JZ      LISTOUT         ;...TRAP DISABLED


;Get to here if F/F is being sent to listing
;AND form feed trap is active.
;Since we are already in a BDOS function call we
;should NOT loop back into another BDOS function call.
;Do console I/O directly thru this USER area.

;Operating program in TPA should have at least 10 stack bytes
;or else a new stack should be set up after saving
;the current stack.

        PUSH H! PUSH D! PUSH B!
        LXI     H,WAITMSG       ;First show the wait message
LISTO   MOV     A,M
        CPI     '$'
        JZ      LIST1
        MOV     C,M
        CALL    UCONOUT
        INX     H
        JMP     LISTO
LIST1   CALL    UCONIN
        CPI     CR              ;Wait for a C/R from console
        JNZ     LIST1
        MVI     C,4             ;Delete wait message
        CALL    UCONOUT
        POP     B! POP  D!  POP H
;Restore old stack if new stack was assigned
        MVI     C,NULL          ;Send null to list device


LISTOUT:
        IN      STATUS          ;Get TTY status
```

```
        ANI     TTYMASK         ;Check for BUSY
        JZ      ULIST           ;Wait until not busy
        MOV     A,C
        OUT     TTYDATA
        RET

ULISTST:

        RET                     ;THIS FEATURE NOT IMPLEMENTED
;
UPUNCH  CALL    LIST            ;First put it on the printer
        JMP     UCONOUT         ;NOW put it on the console


WINITR:
        ;Any warm boot initialization goes here.
        PUSH    D
        PUSH    B
        MVI     A,ESCAPE        ;First character for disabling
        OUT     TTYDATA         ;..OUT OF PAPER error
        MVI     A,VIOBYTE
        STA     VIOFLAG         ;VIO-C to lower case
        LXI     D,HITHERE       ;Point to warm boot message
        MVI     C,09
        CALL    BDOS            ;Show warm boot message

        MVI     A,'8'           ;Disable OUT OF PAPER error
        OUT     TTYDATA
        POP     B
        POP     D
        RET

CINITR:
        ;Hardware initialization on cold boot goes
        ;here if needed.  Make sure it ends with a RET.

        CALL    CLEAR           ;VIO-C is U/C mode now
        MVI     A,VIOBYTE       ;Set the VIO-C to U/L
        STA     VIOFLAG         ;   case mode
        XRA     A
        STA     IOBYT           ;and store.
        RET

;Send a single line warm boot message if form feed trap is OFF
;Send two lines of warm boot message if form feed trap is ON
;
HITHERE DB      13,10,10,'HEAD QUARTERS - Corvallis, Oregon 97330',13,10

FORMBYT DB      '$'
        DB      '  (Trap all Form Feeds)',13,10
        DB      10,'$'


;Form feed has been trapped message
WAITMSG DB      13,10,'FORM FEED TRAPPED - PRESS RETURN TO CONTINUE $'

;
USRLEN  EQU     $-CINIT         ;Length of USER AREA
```

End Listing

Add Direct Video Output to Sinclair ZX81

by Jerry Chamkis

Jerry Chamkis, Box 18093, Austin, TX 78760.

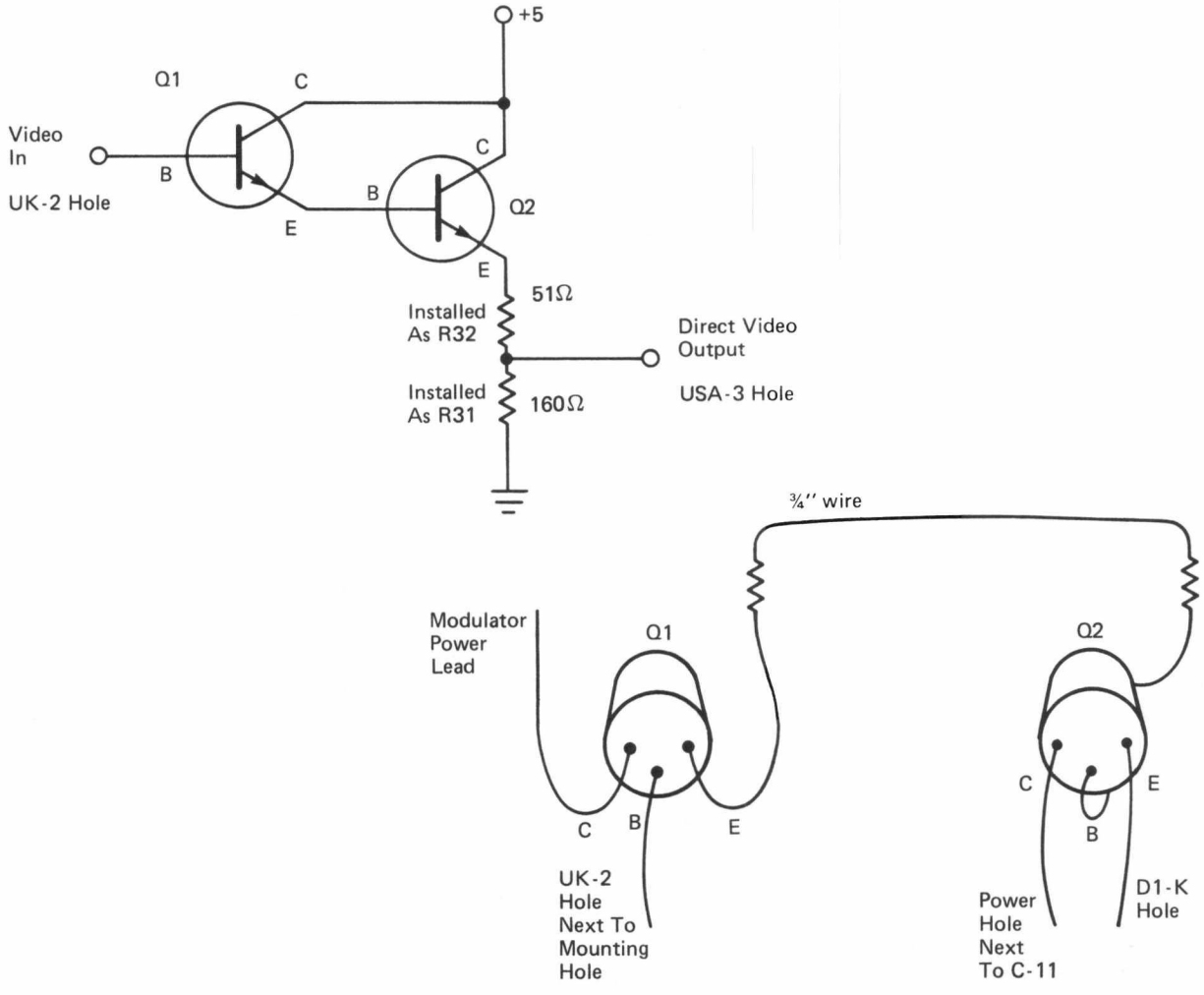
Here's a simple modification you can make to your Sinclair ZX80 or 81 computer to add a direct video output. This allows you to use a composite input video monitor instead of a television set as your display device. If you have a ZX81 in the United States with a UHF modulator, you can mount all of the parts in existing holes on the printed circuit board (provided for components used only with the European video standards).

None of the component values are critical. Transistors Q1 and Q2 may be any small signal high-frequency NPN device, but it is preferable to use plastic-encased transistors since some of the leads must be bent alongside the case. The values of R1 and R2 may be varied to suit parts available or to adjust the video output level. I soldered a phono-type jack on the side of the modulator to serve as an output connector but the signal could just as well be led out on a piece of cable.

One interesting use of the direct video output is to separate out the sync signals and regenerate the horizontal sync with a phase locked loop. When the PLL detects a loss of horizontal sync (ZX81 in the FAST mode or ZX80 always) it triggers a one-shot and holds the video blanked for a completely jitter-free display in all modes.

DDJ

ZX81 Direct Video Modification



Using Computer Bulletin Boards

by Marlin Ouverson

It has been brought to our attention at *DDJ* that a fair share of even our sophisticated readers have no experience with electronic message systems. This article will help anyone get started – the many benefits which accrue will become apparent after only a few hours of experimentation on various systems.

How to Use Typical Systems

Tapping into the current of readily available information is simple. The People's Message Service (PMS) software is exemplary. Originated by Bill Blue, there are now many PMS systems operated independently. They are easy to learn to use, and each has a list of several hundred other systems across the United States. A list of telephone numbers which call up PMS systems is shown in Figure One.

Any terminal can access PMS and similar systems, regardless of the "host" system's configuration; as can any computer running terminal/communications software. Such software ranges from "dumb" to varying degrees of "smart." The most rudimentary use your CRT as a display; keystrokes go through the RS232 port to the CBBS, which echoes them back to your CRT. More powerful communications programs enable better use of your own computer's resources, such as scrolling through several pages of text, file saving, and transmission of preformatted files.

Step One

Make certain a modem is connected to your computer. Acoustic couplers are the type with rubber receptables into which the telephone handset fits snugly. Direct-connect modems use the modular phone jack to plug directly into the phone, eliminating possible room noise.

Step Two

Dial the phone number of a computer bulletin board. If not busy, the system should answer within two or three rings and begin sending a carrier signal. When you hear that tone, place the handset into the coupler (or if using a direct-connect modem, depress the DATA switch).

Step Three

If nothing appears on your CRT after Step Two, then hit your Return or Enter key a couple of times. Some computer bulletin boards require you to know this, while others will send a prompt to the screen telling you to do so. It is used to determine your baud rate.

Step Four

You are now communicating with the host software. You may be asked to enter a password; or if you are a new user your name and location will be requested. New users are usually given special prompts and help messages. At some point, you will encounter a menu of commands available to you. They enable you to use the system's data base; the value of what you retrieve from it can be seen as a function of (1) the information on that particular system, and (2) how well you learn to use the commands for entering, manipulating and retrieving the information.

Step Five

Have at it! Try using commands to scan message "headers" for items of interest to you. Then retrieve those messages to read them. Can't get at one? It is probably encoded with a password for a particular individual's eyes only. Try entering a message of your own – tell other users you have been online, where you are from, anything you'd like to share. It could generate responses, which will be on file the next time you logon.

Step Six

It's easy to spend hours experimenting, getting to know a system. If it is a good one, chances are that many others are waiting to use it. Some CBBS's limit the number of minutes per day for each caller, while for others it is an honor system. For most systems you should type G (for goodbye) to logoff, or something similar listed in the Command Summary.

When you logoff, most systems will ask if you have any comments for the Sys(tem)Op(erator). If not, you are disconnected – but remember to hang up the telephone, in case Ma Bell thinks you are still using her services! It doesn't hurt to be sure. . .

Step Seven

Get used to these simple steps; use several different systems. Discover how the hierarchical structure of Conference Tree systems differs from more linear CBBS's. Find out how to use electronic bulletin boards with electronic mail features, try logging on to systems maintained by special interest groups. You can experiment at beginning an ongoing dialog with users who may share some concerns you have – perhaps it will spin off into a CBBS of its own.

When you have a mild proficiency on one of these systems, find out if it has an expert user mode. Not only will it be faster to operate, there may be additional commands which will increase the power of the system for you and multiply the uses you find for telecommunications.

Conclusion

This has been an intentionally simplistic approach to computer bulletin boards, written to help those who feel they have been left behind in the dust, or to inspire any who haven't understood the importance of the underlying concepts. *DDJ* has always had a strong interest in telecommunications; our company sponsors research and development by PCNET (Personal Computing NETwork) into communications protocols. For electronic discussion of the subject, they maintain a CBBS at 415-863-4703.

We highly encourage any and all to investigate what electronic communications has to offer. Watch *DDJ* for articles about readers' experiences and ideas. Possibly more than any other application of computer technology, this field is open to innovators to shape how computers will serve our world. The technical and social implications are many and far-reaching; we hope every reader will participate. DDJ

Figure One

Numbers of selected People's Message Service systems. These systems are relatively easy for newcomers to master, and contain a list of other bulletin board systems. If none of those listed are in your area code, call the nearest one and use its "O" (for Other CBBS's) function to locate one within your local calling area.

213-334-7614	**317-787-5486**	**509-943-6502**	**714-561-7277**
213-346-1849	**408-370-0874**	**612-929-6699**	**714-582-9557**
216-867-7463	**415-493-7691**	**617-767-1303**	**714-772-8868**
312-295-6926	**415-851-3453**	**713-233-7943**	**913-341-3502**
312-373-8057			

Figure 2.

A sample session on a PMS system.

```
PMS - PALO ALTO, CA.

USERID (N=NONE) :
LOGGING CALLER #25328 AT 11:36:29 PST

Type "N" for system news.
Type "?" for commands.

Command ??                                   ← user typed "?"

----->> System Commands

E  = Enter a message into system.
F  = Features, articles, excerpts.
G  = Goodbye. Leave system (hangup).
H  = Help with various functions.
I  = Information about system.
K  = Kill a message from the files.
M  = Message alert. Messages for you?
N  = News -- System news.
O  = Other systems current summary.
Q  = Quickscan of message headers.
R  = Retrieve a message from the files
     (may be used during a scan to flag msgs).
S  = Scan of message headers (add - to end of
     number for a reverse scan).
SR = Selective message retrieval
     (X aborts, * resumes).
T  = Time, date and connect time.
U  = User modifiable system function.
X  = eXpert user mode (toggles on/off).
Z  = Continue message entry after abort
  = Prints list of commands.
*  = Retrieve messages flagged during scan.
TEST  = Modem continuous test loop.
NEWCALL  = Information for new callers.

Command ?S                                   ← user chooses "S"

Start at msg# (5605/9225) ?9000              ← start scan at #9000

Msg# 9003 on 02/24/82 @16:22 (2) <Pvt>
Subj: HAHA!!, To: MARTY TERMAN
From: Danny Wells

Msg# 9005 on 02/24/82 @17:27 (16) <Pvt>
Subj: R/ DECODER, To: Danny Wells
From: DAN DRUKER

Msg# 9006 on 02/24/82 @17:43 (8) <Pvt>
Subj: MISC..., To: SYSOP
From: DAN DRUKER

Msg# 9008 on 02/25/82 @11:42 (11)
Subj: r/hmb, etc..., To: ken stuart
From: Ron Carpenter

Msg# 9020 on 02/25/82 @22:10 (16)
Subj: for sale, To: all
From: Chuck George

Msg# 9035 on 02/27/82 @01:12 (4)
Subj: HP 2000 MANUALS FOR SALE, To: ALL
From: John Feibusch

Msg# 9042 on 02/27/82 @15:34 (4)
Subj: MODEM MAN NEEDED, To: ALL
From: HOWARD H SMITH

Msg# 9044 on 02/27/82 @19:28 (6)
Subj: CP/M ON MICOM WP -- QUESTION, To: ALL
From: Philip Casella

Command ?R                                   ← user chooses "R"

Msg# (5605/9225) ?9006                       ← want to read #9006

Msg# 9006 on 02/24/82 @17:43 (8) <Pvt>       ← password protected
Subj: MISC..., To: SYSOP
From: DAN DRUKER, LOS ALTOS,CA

Password ?I  don't know it...

Msg# (5605/9225) ?9042                       ← user selects another msg.

Msg# 9042 on 02/27/82 @15:34 (4)
Subj: MODEM MAN NEEDED, To: ALL
From: HOWARD H SMITH, PALO ALTO

GTE LENKURT,SAN CARLOS,(WHERE I WORK) NEEDS AN ENGINEER
INTERESTED IN HIGH SPEED MODEMS. CALL ME 415-595-3000 X2747
(WORK) OR 415-494-2915 (HOME) ANYTIME, IF YOUR INTERESTED.
THANKS

Command ?g                                   ← logging off

Comments/suggestions (y/n/r) ?n

Connect time 03:01

It's 11:39:30   Goodbye..
```

End

Software Review

COMMWHIZ

A Smart Terminal Program for TRS-80's

COMMWHIZ
Smart Terminal Program for TRS-80
Models I and III
(one disk drive required)
$79.95 from
VolksMicro Computer Systems, Inc.
Circle No. 302 on Reader Service Card
Reviewed by PCC Staff

Everyone and his brother who sells communications software these days calls it "smart" – even if the only thing with smarts is the price. Thankfully, there are a few exceptions to the rule, one of which is COMMWHIZ. When the Creator was handing out software IQ's, this package received nearly Mensa-class intelligence.

In this instance, "intelligence" actually means there are many options. If you are used to dumb terminal software, you know the feeling of scribbling madly to copy a piece of data before it scrolls off the screen. Slightly less dumb programs allow one to freeze the action for more leisurely note taking – at a stiff price if you are on a commercial time-sharing system.

The folks who designed COMMWHIZ themselves spend a lot of time on private and public data base/CBBS systems. The have written a software package which allows the user to enter a more powerful command mode. This provides features (see Figure 1) proven valuable by experience, not just in theory. Additionally, it is menu driven, practically eliminating the need to refer to the reference card.

Here are a few of the things it allows one to do:

- Send one or two pre-defined user messages (such as a logon code and a password) with only two keystrokes. This handy feature helps ensure error-free, first-try logon.
- Call up the directory of disk files, useful for locating a file to be transmitted and for avoiding duplication of file names.
- Create a buffer in RAM for storage of material received or to be transmitted. 48K systems will have about 30K of buffer space, a generous amount for text messages, and adequate for uploading/downloading of software.
- Fast command to toggle the buffer open or closed, or the printer on or off. This helps reduce clutter caused by saving unneeded information or filling disks to overflow with unedited material.
- Execute certain DOS commands without leaving terminal mode. This is a handy function, but must be used with care – see comments below.
- Auto open and close of the buffer. This permits other computer hobbyists using COMMWHIZ, or some computer bulletin board system (CBBS) with downloading features like those of FORUM-80, to open your buffer before commencing with file transfer and to close it afterward. Used properly, this will help prevent programs from becoming contaminated with direct statements from the host communications software.
- View buffer and check remaining memory. You can look to see if the buffer has contents, whether you captured all the desired data, and how much space is left in the buffer or on disk.
- Use Scripsit, Electric Pencil or other word processing software to prepare text in advance of logon, or to edit files received. This can save valuable on-line time and, of course, utilize the best tools for text preparation, partly bypassing the need to use any host system's less-sophisticated editor.
- Other commands include open and close buffer, format screen width, initialize serial port, load file to buffer or buffer to disk, modify source file (EDTASM to ASCII and vice versa), return to DOS without reset, transmit buffer and toggle between half- and full-duplex mode.

Additionally, entering a special mode by simply pressing the up-arrow key enables one to generate the characters shown in Figure 2. These are particularly helpful when using remote systems which may require one of these symbols in order to logon or to use sub-menus of commands; or for just getting fancy. Our copy of COMMWHIZ interpreted the underline option as a backspace, although the maker assures us this glitch must be unique to us. . .

The Program in Action

The user is first presented the opportunity to input up to two pre-defined messages. Following that, a status line appears at the top of the screen (as in Figure 3). The user could then, if he chose, access a CBBS without learning any more about the software. Anyone can use the program immediately and explore its more powerful features while in use.

If he has dialed a remote system over the telephone and is required to enter a password or other code, he could have pre-defined this when booting the COMMWHIZ disk. When prompted for it by the host, he may enter Command mode by pressing Shift-Up-arrow and then pressing 1. His pre-defined message is sent, but not displayed on the screen (protecting his password from onlookers).

Whenever the user executes an option from the Command menu, he is returned to terminal mode. It is possible to miss transmissions from the host if Command mode was entered during incoming data instead of waiting for a prompt, but unless the user is hesitant or unsure, he is returned to terminal status within two or three seconds.

CBBS's have different methods for entering text messages, so COMMWHIZ allows for different methods of transmitting text (see Figure 4). Usually, one inputs a single line at a time, followed by a carriage-return line-feed. This means that although one may have prepared text in advance with a word processor, he must transmit it a line at a time, waiting for the host's prompts at the beginning of each line. COMMWHIZ allows one to define such prompts before transmission; it will then relay the entire message, waiting after each line for the proper prompt before proceeding.

While in our office test mode (echoing our messages to the CRT without going through a host) we discovered a possible sore point. A small portion of the sub-menu related to Transmit options is spliced into very short (2- to 3-line) text messages. Evidently an unspecified minimum of space on the display is sent – longer messages must write over that area and so do not include the system information therein. This small bug never once interfered in our actual on-line tests.

A feature has since been added (not available on our review disk) which uses a timing loop to pause between lines. This circumvents the need to know the specific system's prompts in advance. It may also be useful when the prompts for message headers are different from those for the text. There are possible draw-

backs if this feature relies solely on a time factor, so it should be used discriminately until it proves reliable.

DOS commands can be used while in COMMWHIZ. If you have an important buffer-full of information which needs to be saved, but your diskette is full and no spares are at hand, you can KILL an existing file to make room. RENAME is occasionally handy, as well. Although the the documentation advises strongly against use of FORMAT or BACKUP in its mode, it says "most DOS commands are useable . . ." Our experience did not show this to be true; or perhaps too literally true.

In several instance, attempts to use a DOS command returned us to TRSDOS, folding up the buffer and crashing the program. We were able to re-boot and catch our connection to the remote CBBS's without having to logon again, a minor consolation. The fault is not entirely with the terminal software: such crashes usually resulted from improper use of DOS commands which required system error messages. Some, however, should be forgivable errors, especially accidental use of lower-case letters.

Once a session has ended and the telephone is back in its cradle, the user is still in terminal mode. If he has material in the buffer to store on disk, he enters Command mode and presses the letter S. This dumps the entire buffer contents onto disk: line feeds, control codes, and errata ad nauseum. Most conservationist users will then want to edit those files of extranea. Any good editor or word processor should make relatively short work of this task.

A small irritation appeared during preparation of this review. Each text line stored ends with a carriage return. Scripsit interprets that code correctly and displays it as a small block. Each line begins with a right square bracket; that is probably the host system's code for a line feed. Now, in Scripsit a right bracket means "end-of-text-block." But the global delete function won't remove either of the little buggers, so every single line (that must be cosmetically perfect) must have those symbols manually deleted, instead of using the convenient repeat command otherwise available in Scripsit.

The makers of COMMWHIZ tell us this crops up occasionally. Such embedded characters are evidently a function of the host system from which text was received. Scripsit doesn't print such text markers unless specifically commanded, and other editors or word processors might easily search through and strip off the first character of each line, so this is a minor quibble at best.

An interesting note: since one can leave COMMWHIZ, enter DOS and load programs without re-booting, we often do just that to load Scripsit for the above purposes. At odd times like subsequent disk I/O, though, the COMMWHIZ cursor will mysteriously appear, putting an uncorrectable hole in the document being prepared.

Documentation

COMMWHIZ is supplied with a 60-page manual. The first half is a tutorial, the final chapter of which deals with advanced features of the system. The last half of the manual consists of sample sessions on various CBBS and commercial data base systems, showing logon, prompts and command summaries. There are also fairly extensive lists of CBBS telephone numbers. This last section will be particularly helpful to people completely new to telecommunications or to the private, no-cost systems mentioned elsewhere in this issue of *DDJ*.

The manual is to the point, contains all the information needed, and is never condescending. It should have more carefully emphasized the danger of opening a closed buffer – its contents are lost. This is mentioned once, but is easy to overlook until it has happened more than once; and once is too often. The last section on sample sessions should have used a different typeface for author comments, which are interspersed closely with the actual printout of the sessions.

Summary

We recommend COMMWHIZ as a communications package which is both powerful and easy to learn. It is useful from the first session. Its creators are responsive, helpful and available. They have included free on the same disk with this program a utility which "permits rapid error-free file transfers between TRS-80's . . . or from the TRS-80's to CP/M systems running . . . XFRCPM80.COM." Many of the suggestions we would make are more deficiencies caused by a lack of standards among CBBS's than by this software.

When entering the Command mode, it would be helpful if the program generated a Control-S (a common pause command) so the user wouldn't miss any incoming data. If the user attempts to toggle on the printer and it is not connected, the "Printer not available" message flashes on and off the screen too quickly to be noticeable, not to say legible. And there should be a safeguard on that buffer: trying to open it when it has contents should prompt a verifying query; trying to toggle it open when it hasn't been "turned on" by the Command mode should show an error message rather than allowing the user to assume he is capturing anything in RAM. The status line should differentiate between a buffer that is actually closed and one that has been temporarily toggled shut.

We have concluded this is a fine and truly "smart" terminal program which we will continue to use. Our experience with the company and people who make it lead us to believe they will continue to improve an already fine product.

DDJ

```
  MODE=Cmd    BUFFER=Clsd   AUTO=OFF   PRTR=OFF   COMM= 300 N81F
.....

      (A)uto open & close              (O)pen buffer for input
      (C)lose buffer                   (R)eturn to DOS
      (D)irectory                      (S)ave buffer to file
      (E)xecute DOS command            (T)ransmit buffer
      (F)ormat screen                  (U)se of memory
      (I)nitialize Serial Port         (V)iew buffer
      (L)oad file to buffer           duple(X)
      (M)odify Source file            COMMAND ==>  .
```

Figure 1. Command mode menu. Not shown are (P)rinter toggle on/off, (1) to send an optional user-defined message, and (2) to send the second user message.

```
   MODE=Spcl   BUFFER=Clsd   AUTO=OFF   PRTR=OFF   COMM= 300 N81F
0=NULL   1=Escape   2={   3=}   4=¦   5=~   6=\   7=@   8=[   9=]   -=_
```

Figure 2. Special character mode options, available by pressing the up-arrow.

```
  MODE=Term   BUFFER=Clsd   AUTO=OFF   PRTR=OFF   COMM= 300 N81F
.....
.
```

Figure 3. The status line at top of screen always displays the current status of the user's system. Any of the information can be changed by using Command mode options.

```
Select transmit option
     1.  Send with no control characters
     2.  Send with Auto Open and Close
     3.  Transmit in Prompt mode
     4.  Return to terminal mode
```

Figure 4. The sub-menu of choices for transmitting text.

BOOK REVIEWS

Interfacing to S-100/IEEE 696 Microcomputers
by Sol Libes and Mark Garetz
OSBORNE/McGraw-Hill
321 pages, $15.00
Circle No. 312 on Reader Service Card
Reviewed by George W. Jolly

A draft of the proposed standard definition of the S-100 bus was published in the July 1979 issue of *Computer* magazine. *Interfacing*, based on that draft, explains how S-100 bus computers work. To understand this book, you need to be familiar with digital logic and schematic diagrams. If you intend to build any of the circuits described, you need experience building circuits. The software examples presented are in 8080 assembly language, but are also generally flowcharted for you.

An S-100 board must do certain things: it must decode address lines to determine whether or not to do something; control lines tell it what to do; data lines pass the information being processed. If the total computer system is to function correctly, all boards must handle these bus wires the same way. The IEEE definition, reprinted as an appendix to the book, attempts to ensure this by spelling out the meaning of the lines in exact language. The authors augment this definition with actual circuit examples.

Although *Interfacing* deals primarily with the S-100 bus interface, it also explains a variety of outside-world applications. For example, Chapter 10 includes several circuits for controlling AC power with on/off signals from an output port of a computer. The circuits shown are simple, making *Interfacing* a good sourcebook in addition to its main function. The authors give simple programs to illustrate the driver software necessary.

Later chapters explain more advanced system concepts. The authors give a concise, plain-language explanation of interrupt processing, and explain the specific methods used by 8080 and Z80 microprocessors. Temporary bus masters (DMA devices) occupy a chapter thick with timing explanations and example circuits. However, the design of the CPU board itself is not included; dynamic memory is also excluded. Debugging circuits such as a breakpoint and a single-stepper are given, as well as a jump-on-reset circuit.

Although *Interfacing* is overall a very good book, it was not as easy to read as most magazine articles. The explanations are good, but more concise than conversational. *Interfacing* is more suited to serious study than for recreational reading. As a sourcebook of specific circuits, or as a text for S-100 concepts, it is excellent.

Lecture Notes in Computer Science
by G. Goos and J. Hartmanis, Eds.
Springer-Verlag
190 pages, $11.80
Circle No. 332 on Reader Service Card
Reviewed by Robert Irving

This paperback is aimed at the professional writer of interactive programs, with two fundamental messages: (1) try to be a little less "professional," get inside the head of the non-programmer who will be using your interactive system and make it understandable to him; and (2) make both documentation and command structure of the interactive system as close to the "natural" language (i.e., English) as is practicable within the system constraints.

A text editor is used as the vehicle for the discussion, both for representing development of an interactive system and a set of user instructions, and as the basis of an experiment in determining those factors which lead to the best understanding (quickest use) and greatest efficiency (fewest errors). Some of the conclusions are reasonably obvious: if you allow abbreviated commands, be consistent. Don't sometimes use the initial letter, other times two letters, and still other times three, without some discernable rule for application. Easy to say, but not so easy to do!

The authors' "Assistant" program uses initial letters for any keyword except three which must be spelled out completely, presumably because these are potentially destructive commands – but one of them is PRESERVE which puts the text into file. One really unique item is a command called UNDO which does just that – it undoes the undesired result of the last executed command. How many times have you wished that you hadn't just deleted everything to the bottom of the text file, having forgotten that you had just referred to the title of the article to check its wording – and consequently dumped the whole text into Limbo? Of course, such a function could use a lot of memory, as in my extreme example – but memory continually becomes cheaper, and human time more expensive.

One last item is worth mentioning: the book ends with a chapter on how to design experiments to ensure that your "improvements" really improve the program.

Introduction to Computer Organization
by Ivan Tomek
Computer Science Press
456 pages
Circle No. 322 on Reader Service Card
Reviewed by George W. Jolly

Introduction to Computer Organization is really two books in one. The first half deals with structure and design of digital circuits; the remainder of the book covers computer architecture on a larger scale. Intended as a college text for a one-semester course, *Introduction* avoids electrical details, focusing on organizational structures common in computer systems. The material is designed to give programmers, who may only know a high-level language like Fortran, an understanding of how computer hardware works. As a survey of important topics, the book is excellent.

The first five chapters explain the basic of digital circuitry. Electrical details are not at issue here, but rather the behavior of functional units such as gates, flip-flops, multiplexors and the like. One section does explain TTL voltage levels and the effect of noise on the circuit, so this subject is not neglected. State diagrams are used to illustrate the function of sequential circuits. These diagrams are explained at the outset.

Chapters five and six form a bridge between the circuit design and the computer architecture sections of the book. These chapters deal with coding, where numbers stand for things, and with arithmetic as used in computers. Considerations for the design of a character code are discussed, and the ASCII and EBCDIC codes serve as illustration. A brief section on error detection explains the use of parity bits, but the more powerful Hamming codes are only mentioned. The treatment of arithmetic is good, and even

includes a discussion of decimal arithmetic in hardware.

The remainder of *Introduction* explains computers. This is a hardware view, full of program counters, memory address registers, and data busses. A tiny CPU is outlined, capable of only a few instructions, and a driving microcode program is developed. Most of the truly nasty problems of real microcode are removed, and the result is a fascinating insight into the workings of a computer. *Introduction* gives an overview of common instructions found in real computers, and explains the stack found on most microprocessors.

Introduction also gives an excellent overview of memory and peripheral devices. Terminals, printers, disks, drums are all explained in basic terms. Good coverage is given to basic methods of handling peripherals with software. The treatment is based on plain English and simple diagrams, so the reader need not know any particular computer language.

To put the subject into better perspective, *Introduction* covers several real computers. The Heathkit Microcomputer Trainer ET-3400 provides an example microprocessor system. The author uses the Motorola 6800 chip for an in-depth explanation of subroutine linkages, interrupts, I/O, and instruction timing. Minicomputers and large computers are outlined for insight into other architectures. The PDP-11 family, the Univac 1100/60, and the CDC Cyber 170 series provide a diverse view. A final section on future trends rounds out the picture nicely.

A workbook is available to accompany this textbook. Throughout the text the reader is referred to the workbook for more examples; however, the text stands well on its own. The preface to the text warns that some problems in the workbook contain important material not covered in the text, and recommends using the two books in parallel. Unfortunately, this reviewer has not seen the workbook, but it should be worth a look or two. Overall, *Introduction* is an excellent text.

DDJ

(Continued from page 30)

simulator is completely independent and can be used with any computer with a serial port for communication with the clock simulator. The advantage of an independent clock is the lack of interference with the host computer and the capacity to implement multiple clocks with little burden on the host's CPU speed. This would not be true if the clocks were implemented on the host computer itself, since the host computer will be slowed down considerably by the extra CPU cycles consumed by the clock simulation. We have found the clock simulator most helpful in performance analysis when we compare the relative efficiency of various computer algorithms and evaluate the relative speeds of commercial interpreters and compilers. The application of the clock simulator in computer performance analysis will be the topic of a future paper. DDJ

(Figures 1-6 begin on page 34)

Hayes Stack Smartmodem
Hayes Microcomputer Products, Inc.
Circle No 322 on Reader Service Card
Reviewed by Rich Henderson-Gragg

The Hayes Stack Smartmodem is a 300 baud, auto-answer/auto-dial originate, FCC part 68 certified modem with standard RS-232 computer interface. As an FCC certified device this modem can be directly connected to the phone line via a standard modular jack. The direct connection facilitates the auto-answer/auto-dial capabilities as well as eliminating the noise problems inherent in acoustic coupling. 12 VAC power for the modem is provided by a UL listed power pack similar to those used by calculators.

The unit is approximately 1.5" x 5.5" x 9.6" which allows it to fit conveniently on or near your computer (fits fine on top of Apple disk drives). The case is part of the Hayes design which allows different products of this series to "stack" together. There is a ridge on the top of each case to provide stability and a standard desk phone can sit firmly on top of the complete stack. The front Panel has 7 indicator leds that provide the user with information on the state of the modem. These lamps are:

- MR – modem ready (power on)
- TR – terminal ready (DTR from terminal or internal switch)
- SD – send data (flashes when data is received by modem from local terminal)
- RD – receive data (flashes when data is sent by modem to local terminal)
- OH – off hook (modem has "picked up" the phone)
- CD – carrier detect (modem has recognized a carrier from the line)
- AA – auto-answer (auto answer mode is enabled)

Installation is quick and easy. Connect your terminal or computer (we used an Apple 3) to the RS-232 connector on the back, plug the modular cable provided into the jack in back and into the telco line jack, plug in the power pack, switch on and you're ready to go.

As this is a "smart" device, you communicate with it through your terminal. When the power is first applied, the modem initializes itself, lights the MR lamp and awaits a <CR> from the terminal to establish the baud rate between it and the terminal. You can now command the modem to originate/answer a call or enter an auto-answer mode with simple command strings such as

ATDT 408,555-1212

This command string is AT (Attention) DT (Dial touchtone) 408 (dial 408) , (wait 2 seconds) dial the rest of the number (the - is ignored). When the string is entered (terminated by a <CR>), the modem will execute the command. Then, if the internal software switch is set, it will activate the monitor speaker so you can hear the progress of the call (very handy if you get a busy signal) and deactivate the speaker when it detects the carrier from the other device. (With the software switch you can control this feature to allow the speaker to come on and stay on, or not to come on at all.) The speaker is listen only, not a speakerphone, therefore if you get a person ("what number did you dial") you will need to pick up the telephone to answer. If the modem doesn't receive carrier in about 30 seconds (this can also be changed) it will hang up and send a "NO CARRIER" message back to your terminal. Also if you hear a busy, sending any character to the modem while it is waiting for the carrier will abort the call with a "NO CARRIER" sent back to you.

A further interesting feature is the ability to deal with older office pbx-type equipment which may require pulse dial to get the telco trunk lines. This command string would handle such a problem,

ATDP9, T555-1212

The DP, would pulse dial the 9 access and wait 2 seconds before going into the touchtone number.

With this system a wide range of commands is possible to handle about every conceivable condition including originate a call but using answer mode tones if calling a friend who has an originate-only modem, return to command state at end-of-line so additional commands can be input for a phone order or bank service. All 12 touchtone keys are provided including * and #. Other commands are used to set internal software/hardware conditions such as redial last number without having to type it in again, auto-answer on nth ring of phone, carrier on/off, monitor speaker on/off, reset and a group of parameters including: character used as <CR>, character used as <Back-Space>, how long to wait on loss of carrier.

The manual is detailed, well-organized and well-written. The introduction tells you which parts of the manual to read to get on the air and which to read to learn more later. The appendices cover RS-232 connections, modular phone wiring diagrams, dial pulse and touchtone frequencies, specifications, use with amateur radio and return for repair procedures (in the very unlikely event you need it). In the back of the manual they have thoughtfully provided a tearout summary card of the commands and functions.

The only real problems we experienced with the system were getting used to the idea that the Smartmodem could take care of the line management and that programs didn't need to worry about answering the phone, checking for carrier or how long to wait on loss of carrier. All the program really needed to do was watch the RS-232 Carrier Detect line to determine if it connected to another device. The only annoyance we discovered was that the commands sent to the Smartmodem had to be uppercase. We spent 10-15 worried minutes trying to figure out why the modem wouldn't acknowledge a command before noticing that the terminal was in lower-case mode. This requirement was implied in the manual as all examples were bold, uppercase characters, but we could find no specific statement to this effect.

We've used the Smartmodem to talk to a variety of computer systems with no trouble at all. While we haven't had the opportunity to test the "banking" features we have no doubt that they will work as described. The system installed easily and worked with no fuss other than remembering to send upper-case commands. The two programs we wrote in Apple 3 Pascal had no problem sending instructions to the modem and functioning with the auto-answer feature. It is an attractive package and design, and should work well with almost any terminal or computer where the RS-232 port is wired as DTE (data terminal equipment). In the event that your computer's RS-232 port is wired as DCE (data communications equipment) the manual describes how to allow correct operation. ■■J

Guest Essay

Prior Censorship

The Deputy Director of the CIA, Admiral Bobby Inman, announced on January 8* that the Administration would seek passage of technical censorship laws unless the scientific community submits to "voluntary censorship." In recent years the government has attempted to censor cryptographic research and nuclear physics. This new attempt includes the fields of computer hardware and software, among others.

What the CIA wants is called *prior censorship*. It means that the CIA reads any article submitted for publication, and decides whether it should be printed. The readers of the technical journals never know which articles have been rejected; they simply disappear.

Our Constitution and legal system were designed to make it hard to impose this kind of tyranny without public consent, so the CIA is attempting to get cooperation through threats like the one cited above rather than through legal means. Our best defense against this is to immediately make a strong public stand against the attempt.

Buy why should we care? The National Security Agency's voluntary review system for cryptographic papers has reviewed 25 papers so far, and passed them all (or so they say). And there *is* some risk that American advances in technology can be used "against us" by other governments.

I think we should care because information is a valuable resource to *us*. When the CIA censors reports on new research and development because they might reach foreign ears, they also withhold the information from us, the American technical community. Imagine where microcomputers would be today if the knowledge gained by the large-computer field had been withheld from publication – we'd still be diddling bits in 4004's, if indeed LSI had ever been invented. The explosive growth of the computer industry has been fueled by the rapid dissemination of new information; we censor that information at our peril.

Besides, our best defense against foreign domination in technical fields is to do a lot of research here in the U.S. If the prerequisite information is not available, and if the results of the research cannot be released, much of the incentive for individuals and companies to do new research simply disappears.

There is also the argument that we, the public, should have at least as much information as our government has, to protect us against abuses of power. The Federal Data Encryption Standard is a good example. The original standard specified a 128-bit key, but the National Security Agency intervened and had it shortened to 56 bits. The original length provided security against anyone, including the NSA; the new length allows the NSA (and anyone else with Cray-sized computers dedicated to decryption) to read anything sent in the code. This deception was revealed in computer trade journals, and the DES has been widely discounted because of it; but how much of this would have been revealed if the CIA was in charge of censoring technical publications?

What can *you* do about this? First, look in your White Pages under "United States Government" for the nearest Federal Information Center. Call them and they'll tell who your Representative and Senators are, and their addresses. Then, write a letter briefly explaining the situation and your viewpoint. Send copies to your Congresspeople, and to your favorite technical publications. Indicate on the letter who is getting copies. Put messages into computer bulletin boards and electronic mail systems – let your friends and co-workers know. Xerox this page and post it at work. If you're active in a technical society, introduce a resolution that prohibits external censorship of its publications. Get it incorporat into the by-laws if possible; make it hard for *any* individual or agency to pressure your organization's management, by making the will of the membership clear.

If you value your right to know the latest technical information – act now. Next month may be too late.

John Gilmore

John Gilmore works as an independent consultant in the San Francisco Bay Area.

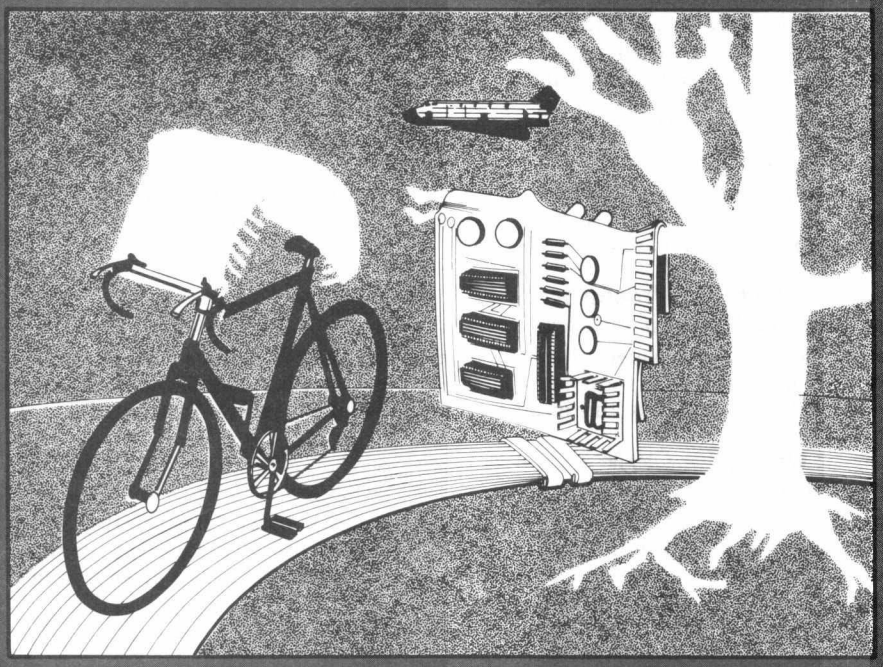
**San Jose Mercury News, 1/8/1982, page F1.*

$2.50
($3.00 in Canada)

DR. DOBB'S JOURNAL

For Users of Small Computer Systems

Number 68 June 1982



Better LET For LLL Basic. . .

. . . SUBMIT to CP/M. . .

. . . CP/M Print Utility

A Multi-68000 Homebrew Project
Al Kossow

Cross-Reference Generator in C
Jeffrey L. Taylor

PDP-1802: A Hardware Simulator
Christopher L. Hamlin

and even more inside!

DDJ T-shirt Contest
Page 61

DR. DOBB'S JOURNAL
For Users of Small Computer Systems

May 1982 Volume 7, Issue 5

Editor – Marlin Ouverson
Assistant Editor – Fritzi Lareau
Contributing Editors –
Dave Caulkins, Dave Cortesi
Marketing Director – Craig Harper
Advertising Sales – Laird Foshay
Circulation Manager – Gloria Romanoff
Circulation Assistants –
Theresa O'Rourke, Terri Marty
Research Assistant – Cynthia Winograd
Art Director – Clifford West
Production Manager – Barbara Ruzgerian
Production Assistant – Joneva Barry
Typesetter – Renny Wiggins
Cover Illustration – Al McCahon

© 1982 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $40 for first class to Canada and Mexico; $58 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers Contributing Subscriber: $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Thomas H. Grohne, G. V. Elkins, Robert M. Connors, William J. McEown, Robert C. Luckey, Robert N. Stabler, DeWitt S. Brown, John B. Palmer, Jack Miller, Burks A. Smith, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Howard E. Decker, Jim Wilson, Frank Lawyer, Chris Pettus, Paul Lutus, Ben Goldfarb, Rodney Black, Steven Weiss, Pat Phelan, John Brodie, James Fleming, Unison World, Christopher Sherbak, Thomas Davis, Jan Steinman. **Sustaining Subscribers:** Gordon Eubanks, Jr., Pete Roberts, Steven Fisher Enterprizes. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors UK & Europe: Hofacker-Verlag, Tegernseer Strass 18, D-8150 Holzkirchen, West Germany. **Asia & Australia:** ASCII Publishing Co., Ltd, Aoyama Building 5F, 5-16-1 Minami Aoyama, Minato-ku, Tokyo 107, Japan.

CONTENTS

Articles

15 PDP-1802 – Hardware Simulator in Assembly Language
by Christopher L. Hamlin
This author chose to execute PDP-8 machine-language programs on a COSMAC 1802. This first of two articles describes the PDP-8 instruction set and front panel functions, and an octal console monitor for the 1802.

22 Multi-68000s in a Personal System
by Al Kossow
If real-time rotation of graphic images sounds good to you then this project, which also supports multi-tasking and several kinds of mass storage, should appeal. Long live homebrew!

26 An Improved LET Processor for LLL Basic
by Norman E. Thagard and George Glenn
The version of Basic widely distributed in earlier years by Lawrence Livermore Lab has an assignment statement which is quite limited. The authors applied compiler theory loosely to this interpreter to come up with a much better LET.

36 A Print Utility for CP/M
by Dan Daetwyler
Tired of seeing wasted printer paper? Would you rather batch your printing jobs than set up SUBMIT files? How about another good use of the ambiguous file name? This utility addresses all these; perhaps you will find ways to make it even more useful.

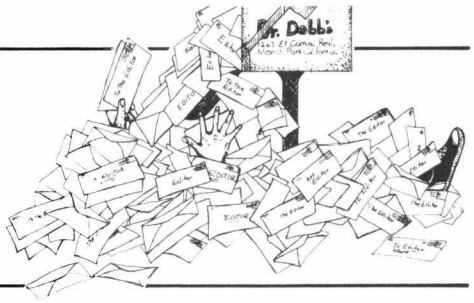
50 A Cross-Reference Generator in C
by Jeffrey L. Taylor
Software conversion is one process which makes you wish you had a cross-reference listing. This one is written for Whitesmith's C compiler and Portable C library, but the author has included aids for adapting it to other compilers and libraries.

Departments

4 Editorial
4 Letters
9 Dr. Dobb's Clinic
12 Of Interest
56 Book Reviews

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, Box E, 1263 El Camino Real, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025 • 415/323-3111

LETTERS



In Sync With Sinclair

Dear *DDJ*:

This letter is a response to the article on the ZX81 which appeared in the April issue of *DDJ*.

I have been using microcomputers for many years and recently purchased a ZX81 with the 16K RAM pack. Let me say that I am extremely impressed with this machine. For $250 one gets a complete computer system capable of being programmed in BASIC or machine language. The 16K memory is quite sufficient for many serious applications including solutions of non-linear equations and other problems which are too big for most handheld calculators/computers. It is certainly a good alternative to a programmable calculator as long as portability is not required.

There are a wide variety of companies producing hardware and software for this machine. While most of the software consists of game programs, the hardware being produced includes additional memory, serial interfaces, modems, sound boards and other interfaces to permit the ZX81 to communicate with the world. Most of the modules are priced under $100 (e.g., a serial interface advertised by BYTE-BACK Co. is $60). Finally, a magazine (*SYNC*) and a newsletter (SYNTAX ZX80) are devoted solely to providing information/programs for the ZX81 and its predecessor, the ZX80.

This machine is certainly going to bring a lot of people into the micro world. I recommend it for educational purposes as well.

Sincerely,
Anthony Skjellum
1695 Shenandoah Road
San Marino, CA 91108

PET Project

Dear Sirs,

I am a student at the University of New Brunswick, Canada, in the senior

Editorial

Improving Reader Interface

Great things are happening at *Dr. Dobb's Journal* these days. With the beginning of summer we have re-dedicated ourselves to publishing more pages, with more articles and more services for our readers. We have always been people-oriented – machines serving humanity rather than the reverse. This has set us aside from the ever more commercial elements, and has spurred us to stay on the cutting edge, a razor-blade position that has earned *DDJ* much respect.

Improving our "reader interface" is something to which we are committed. When you get your next issue, take a close look at the Reader Service card. Beginning in the July issue, you will be able to use it to vote for the most interesting article of the month, as well as to request information about advertisements. The author who (aside from our full-time staff) receives the most positive reader response will get a cash honorarium as a token of reader appreciation. In this manner you will be able to voice your preference in each issue and encourage favorite authors to contribute again.

* * *

Have a look at the *DDJ* announcement on page 61. The first annual Great *DDJ* T-Shirt Contest is on! Long-time friends have been reminding us for quite a while that the three-headed dragon which for years emblazoned the chests of the microcomputing world, has grown a bit tattered. Time has come, they tell us, to bring a new design to life. All readers are invited to quickly submit their ideas, phrases or sketches. The winning entry will be announced in our September issue.

* * *

Now, a note for all of you who have been asking for more of Jonathan Mills' 8080/8086 Cross-Assembler. As you read in his article, the size of the listing is enough to deter all but the most masochistic from attempting to type it in by hand. The author has since arranged for RD Software to distribute the software on disk. It should be available soon; write to them at 1290 Monument Street in Pacific Palisades, CA 90272 and they will hold your letter until specific details are available. Then they will get in touch with you.

* * *

We don't know the American Enterprise Institute, and can't vouch for their ideas or policies. Bruce Schuman recently sent us their motto, though. Whoever they are, their words reproduced below express some things that work well for all of us in microcomputing, and in publishing.

Competition of Ideas

Competition of ideas
is fundamental to a free society.
A free society,
if it is to remain free,
cannot permit itself to be dominated
by one strain of thought.
Public policy derives from the ideas,
speculation and theories of
thoughtful men and women.
Policy makers themselves
rarely originate the concepts
underlying the laws by which
people are governed.
They choose among practical options
to formulate legislation,
governmental directives, regulations
and programs.
If there is no testing
of ideas by competition,
public policy decision may undermine
rather than bolster
the foundations of a free society.

MDO

year and I am taking a thesis project with the Bio-Engineering Institute of the university. The title of my thesis is: "Personal computers adapted for young disabled patients and voice output interface for non-verbal handicapped (text to speech synthesis)."

What I'm intending to do is work with the PET computer and change its normal keyboard to an expanded keyboard (used to accommodate people with motor or limb dysfunctions). Also I will be investigating the possibility of interfacing the PET computer, if possible, with printers other than those compatible with the PET, and also with a speech synthesizer for non-verbal, disabled persons. Lastly, if everything goes as planned, writing some programs for young persons in their first steps of learning.

The reason I am writing to you now is to inquire if you have any relevant material on this subject. If so could you be kind enough to send me information on how I may obtain this.

Thanking you in anticipation.

Constantinos Pattichis
780 Montgomery St., Apt 10A
Fredericton, NB
E3B 2Y1
CANADA

Packing It In

The Editor,

I enjoyed the notes (in "Clinic," *DDJ* #60) on how to convert binary to BCD – without division – by adding hexadecimal "A". About a year ago I was thinking along the same lines, and noted that "A" was part of the opcodes for 8086 instructions AAM and AAD (opcodes D4*0A* and D5*0A*). These instructions convert between decimal (BCD) and binary while at the same time packing a word into a byte/unpacking a byte into a word.

"Is the 'A' in the opcode used as immediate data?" I wondered. I soon discovered that the unspecified machine instructions D410 and D510 convert between an unpacked hex or decimal word (four nibbles 0x0y) and a packed byte xy (i.e., no change in base).

Yours sincerely,
Keith S. Wilkinson
CPO Box 1748
Tokyo, JAPAN 100-91

Doubly-Dense Pascal

Dear Dr. Dobbs,

Subject: Difficulties on getting appropriate hardware & software to run application written in Pascal needing more disk space than 8" single-density.

I have run into the following dilemma in choosing an appropriate machine for a potential client:

1. get machine already having UCSD Pascal for double-density, etc. (such as Altos), but wind up having only 2 serial ports (insufficient for console, printer and modem); also, one Altos user told me the company won't let him have the BIOS source, saying it's "proprietary," making it difficult to customize the printer routine; or

2. get machine (such as S-100) for which hardware can be added at will and wind up having to do considerable programming to make d.d. work for Pascal under a CP/M version of UCSD by modifying the BIOS to handle double-density – must be redone for every disk controller;

3. I have considered rewriting the interpreter to use the disk tables in a CP/M 2 BIOS, effectively making a "ready-to-run" system for d.d. on all CP/M-type machines, but it appears such a system couldn't be legally sold to a customer; or

4. even make UCSD work at all if only one of the so-called adaptable systems which are not CP/M-compatible can be obtained for the customer – which, I believe requires rewriting the entire BIOS for the JMP vectors needed for that "adaptable Pascal."

Other problems: A local computer store didn't know of any backup for the hard disk on Apple (other than 35 minidiskettes or another hard disk!), and I don't know what the story is with other machines – I consider lack of backup to be totally unsatisfactory.

It appears that it will take considerable letter writing and/or phone calls to get the needed facts in order to determine the best solution, and I may have difficulty getting people to understand the dilemma and give me the needed information.

Several so-called data-base systems for micros I've had experience with have limitations that hamper "relatively" sophisticated applications: DBASE2 doesn't delete entries from an index when records are deleted and makes indexing of more than one field in a record extremely difficult; MDBS requires you to write your own editor; SELECTOR-3 takes 15 minutes to delete one record from a 1000-record file. However, I've been able to write such an application from scratch purely in Pascal by using very good and imaginative organization to make it configurable – but the above hardware problems get in the way. All of this may be an argument for dropping UCSD in favor of a CP/M Pascal – losing UCSD's portability advantage.

Sincerely,
Thomas R. Amoth
2215 NW 16th
Corvallis, OR 97330

In Search Of . . .

Dear Sirs:

I have recently come up against certain shortcomings in the packages that I currently use for professional Z-80 software development, and hope that one or more of your readers will be able to offer a solution. Failing that, perhaps one of the software houses will be prompted to respond to my need, which certainly may be a common one.

The applications program which I am developing is composed of 23 relocatable modules, plus a definitions module composed of EQUates, and a COMMON data area. This comes to about 19,000 lines of code and comments. The definitions module and the COMMON area are INCLUDEd with each module assembly, for reasons I will describe later. The total program occupies about 32K code, with another 28K of tables and buffers, and there are 123 globally defined symbols.

Each module is written in Z80 assembly language, for assembly by the CROMEMCO Z80 Macro Assembler, Version 3.07, under CP/M 2.2. The Assembler interfaces to CP/M through a small routine attached to the Assembler, causing CDOS System Calls to be translated to CP/M BDOS Calls. The Assembler generates relocatable code in Microsoft REL format, of course. I greatly prefer the CROMEMCO Assembler over Microsoft's Macro-80 when it comes to appearance of assembly and cross-reference listings. The CROMEMCO Assembler also provides the STRUCT and GLOBAL pseudo-ops which Macro-80 lacks.

The modules are linked together by the Digital Research PL/I linker, LINK-80 Version 1.3, which produces a .COM file quickly and easily in about 4 minutes. It is not possible to use Microsoft's Link-80 Version 3.43 without running out of memory in my 53K transient area.

For debugging, I use the CROMEM-

(Continued on page 55)

DR. DOBB'S CLINIC

by D.E. Cortesi

The Answer, We Submit . . .

We've had our hands on an IBM Personal Computer for a few days now. It's just a computer, after all. However, it has a couple of nice touches in its batch facility (the equivalent of CP/M Submit processing). One is the PAUSE command, which waits for any character to be typed. If the character received is a control-Break (control-c), the batch file is cancelled; otherwise it continues.

That's a handy little function. You can stick a PAUSE command at the point in a batch file where the operator is supposed to change diskettes. Or you could put one between a compile and a link so the job could be killed if there were compile errors.

In fact, it's so handy that we wrote one for use on our CP/M system (Listing 1). One little-known sentence in the CP/M documentation explains that the proper way of aborting a submitted job is to erase the file $$$.SUB. That's what our PAUSE does if the character typed is a control-c.

Of course, you might be deep in a *DDJ* article when the job gets to the PAUSE and so not notice that the system was waiting. For those occasions, it's nice to have the BEEP command in Listing 2. Then you can build submitted jobs along these lines:

```
. . . lengthy commands . . .
BEEP HEADS UP, DUMMY!
PAUSE OK TO GO ON? CHANGE DISKS IF SO.
. . . rest of job . . .
```

With a couple of changes, BEEP becomes FORMFEED, a command to eject the printer forms during the batch job.

Submit We Must

We continue our campaign to raise the world's submit-consciousness with an example. Lots of people use the disk cataloging system written by Ward Christiansen and distributed on the CP/M User Group's disk number 40. In order to use it, you have to give each diskette a permanent, three-digit number, and a name that is descriptive of its present use.

Disks change their uses a lot. Often, we retire one of ours back to the box of free diskettes to await re-use. At that time, all entries for files on that disk should be pulled out of the master catalog, and the diskette should be re-entered to the catalog as "-FREE.*nnn*." The submit file in Listing 3 will do exactly that.

The first thing it does is to list the catalog entry for the disk under its present name. Then it pauses, in case the user has second thoughts. If not, it uses ED as a utility to expunge every catalog entry that refers to the disk. ED is called again to build a one-entry file of catalog transactions to enter the disk under its new name of "FREE." The UCAT utility updates the master catalog.

. . . But IF We Submit . . .

One of the great oversights in Dr. Kildall's little system design was the omission of any convention for command return codes. A command should be able to terminate with a success/failure flag, and it should be possible to test the last code in a submitted job. (The IBM DOS has the same glaring omission, by the way.)

But there are other things that we'd like to test during the course of a batch job. We've seen cases where it would be useful to test (1) whether a file exists, and abort if it doesn't, (2) whether an operand is an ambiguous file reference, and abort if it is, and (3) whether an operand was omitted, and abort the job if it was.

The PAUSE command above demonstrates a general pattern on which such tests could be built. Does anyone feel like writing the IFEXISTS, IFAMBIG, and IFNULL commands for use in submitted files? Please do! Send them in, along with any similar sort of submit utilities you can think of. We've heard there is a simple way of chaining to another command. How about a command that will execute another command only on a certain condition?

A Faster Warm Start

We now have a 96KB CP/M system. The extra 32KB is accessed by setting the extended address lines of the S-100 bus through a Godbout Memory Manager board. When active, it replaces the middle 32KB of the address space, from 4000h to BFFFh. What can you do with an extra 32KB of storage? Well, the only thing we've done with it so far is to cut the time our system takes to do a warm start.

At the end of almost every CP/M command, the system selects the A-drive, seeks to track zero, and reads two tracks worth of CCP and BDOS. That refreshes the code of those programs in case the previous command had overlaid them. Then the BDOS causes a seek to track two and reads the disk directory.

On our system, a warm start used to take just over two seconds – but not any more. Now, when the system comes up in a cold start, the BIOS switches in the extra 32K board and does a block copy of the BDOS and CCP. The extra board is then switched out and the regular storage switched in.

On a warm start, the system no longer loads the monitor from disk. It just switches in the extra RAM and does a block copy from it to refresh the BDOS and CCP. That lowered the warm start time by six-tenths of a second. It also produced two bonuses. First, the BIOS got smaller, since it took quite a bit of code to handle the warm-start I/O. Second, we can now put any disk at all in the A-drive. The A-disk no longer has to be "bootable," except when the power is first switched on.

One other change cut another seven-tenths of a second, but it is specific to our system. It uses the California Computer Systems BIOS, and that code supports all sorts of disk densities. When it selects any drive for the first time after a warm start, it has to feel out the density of the disk. It used to start by reading at single density. Most of our disks are double-density. The disk controller chip would try valiantly to read the disk at single density, not giving up until it had seen the index hole pass five times. Then the BIOS would try again at double density and succeed. We changed the logic so that the BIOS tries all disks first at double density. That cut five disk rotation times from every select of a new drive, including the select of the A-drive that follows a warm start.

The effect of these two changes was to bring the warm start time below one second. The subjective effect is of a *much* snappier system.

A Neater CCP

We have in hand a copy of NEWCCP, a replacement for the Console Command Processor of CP/M 2.2 from MRZ Data

Systems (POB 2571, Warminster, PA 18974). NEWCCP comes as assembly source code for both the MAC and MACRO-80 assemblers. It contains a number of added built-in commands, and improvements on the standard ones. We haven't actually installed it yet, but just reading it and thinking about all the *other* improvements that we could stick into our very own CCP is almost enough to justify the $25 price. If it works when we try it – and it looks as if it should – it will be a real bargain.

Cheap Thrills

That brings up the subject of good, cheap, software. There are a number of really useful, really cheap programs on the market. All too often, the companies that sell these bargains can't afford the price of mass-market advertising, so the products go unnoticed in the general hullaballoo.

If you have discovered a software bargain, send us a one-paragraph review and description (like the one of NEWCCP, above). We'll print it under the "Cheap Thrills" heading. A genuine cheap thrill would be a reliable piece of software that performed a generally useful function, and which came at a remarkably good price.

On the Fast Track

The idea of doing whole-track buffering of diskette I/O has been around for a while. The notion is that, whenever a program requests a read or write of a single sector, the system should in fact read the whole track that contains that sector. The advantage is that reading a whole track at once takes at most two revolutions of the diskette, and usually less. Reading one sector at a time can take two revolutions per sector in the worst case.

There was an early implementation of this idea, for CP/M 1.4 and limited to a Tarbell disk controller, on one of the early CP/M User Group diskettes. Nowadays there are several commercial implementations of the idea that go under various trademarks.

If you want a good, practical discussion of how whole-track buffering is done, you can find it in an addendum being distributed with the CP/M-86 manual set from Digital Research. It's a paper written by John Pierce which gives the theory and contains the skeletal code of a CP/M-86 BIOS to implement the idea. We don't know if DR will send you the paper alone as an application note, or whether you have to order the whole manual set to get it.

Intel Needs Help

From the January-February issue of *Solutions*, a public-relations magazine put out by Intel Corp., we learn that Intel has prepared a Support Library for their 8087 numeric co-processor. It "consists of the function library, a decimal conversion module, . . . and a full 8087 software emulator for debugging without the 8087 component. The library . . . supports the proposed IEEE Floating Point Standard."

Well, that's a lot of software (even if the 8087 does do most of the work), so we suppose that Intel's asking price of $1250 (gulp!) is justified. But is the 8087 so hard to work with that you need twelve hundred dollars worth of interface code? Surely not! Obviously, Intel doesn't know how to sell chips. Let's show them how it's done: let's put some 8087 knowledge into the public domain so that experimenters can use the device. Any readers working with it? What kind of programming does it take?

Promises, Promises

Next month we will return to the problem of making a BASIC interpreter branch faster. Several readers have contributed to the discussion, but none has reported actual measurements or traces of operating programs. It ought to be especially easy to get an execution trace from any Microsoft BASIC; all you have to do is find a way to direct the output of TRON to a file for later analysis. Still, the important question of the dynamic behavior of practical BASIC programs remains open. Computer science is one of the few sciences that can be truly exact, you know. But without measurement and experiment, it isn't a science at all! ▶▶J

Listing 1

```
TPA     equ     0100h
Boot    equ     0000h
Bdos    equ     0005h
CharIn  equ     1
Erase   equ     19
ControlC equ    03h
;
Pause   org     TPA
        mvi     c,CharIn
        call    Bdos    ; get one character
        cpi     controlC
        jnz     Boot    ; just warm-start,
        mvi     c,Erase ; ..unless control-c
        lxi     d,SubFile
        call    Bdos    ; ..then erase $$$.SUB
        jmp     Boot
;
SubFile db      1,'$$$     SUB',0,0,0,0
;               rest of FCB is irrelevant
        end     Pause
```

Listing 2

```
TPA     equ     0100h
Boot    equ     0000h
Bdos    equ     0005h
CharOut equ     2
Bell    equ     07h
;
Beep    org     TPA
        mvi     c,CharOut
        mvi     e,Bell
        call    Bdos
        jmp     Boot
;
        end     Beep
```

Listing 3

```
A>type freedisk.sub

; going to make the disk -$1.$2 a free disk
cat -*.* $1.$2
pause -- does that look right?
xsub
ed mast.cat
#a
#ms,$1.$2^Z0L1K-1L
e
era names.sub
ed names.sub
I-FREE.$2
e
ucat
```

OF INTEREST

by Michael Wiesenberg

For further information on any of the items mentioned in this column, circle the designated number on the Reader Service card and return it to us. Your request will be processed promptly.

Communication Is What Computers Do With Each Other

LogOn, from Ferox Microsystems is a software package that turns your desktop computer into an auto-dial, auto-answer, intelligent terminal that intercommunicates with other computers. With menus, users send and receive text or binary files, exchanging messages, electronic mail, programs, graphic images, etc., at 30 to 120 cps. A permanent directory of frequently-called numbers and computer configurations can be constructed. Then the user selects a computer from a menu, and LogOn dials, redials automatically if necessary, transmits data, and retransmits blocks when necessary (based on CRC checking). LogOn is written in UCSD Pascal, and can be ported to any other UCSD Pascal-based micro, mini, or mainframe, making it upward-compatible. Cost is $150 on 5¼- or 8-inch disk for most systems. (Reader Service **No. 303**)

Tickerscreen, a computerized stock market bulletin board, is a free service for investors with personal computers. Offered by Max Ule & Company of New York, Tickerscreen can be accessed for the cost of a local call in over 150 cities. The service provides closing New York Stock Exchange prices, closing market indexes, commission computation on any stock or option transaction, direct order entry for Ule's clients, and (here's the pitch) a demonstration of Tickertec, a service that displays the NYSE or AMEX tickertape without a fifteen-minute delay. (Reader Service **No. 313**).

Hardware Not Sold in Hardware Stores . . .

The **Rockwell AIM-65** is a true microcomputer-in-a-briefcase. It has its own 120-line-per-minute, 20-column thermal printer, a 20-column alphanumeric display, a keyboard, 1 or 4K RAM and 4 or 16K ROM. Memory expansion to 32K is possible. Applications are installed in ROMs or EPROMs. Software options include BASIC, an assembler, a compiler, Forth, a math package, and Pascal. The stripped-down, fully-expandable machine costs $419, while the complete AIM-65 in a thin briefcase, with power supply, is $518. (Reader Service **No. 323**)

The Micro-Professor is a computer-in-a-book, a Z-80-based single-board computer with 2 to 4K bytes of RAM, and 2 to 8K of ROM. It has a built-in six-digit LED, and a 36-key keyboard. A 3.5-by-1.3-inch wire-wrap area is used for breadboarding. It comes with a built-in speaker and cassette tape interface, and costs $149. (Reader Service **No. 333**)

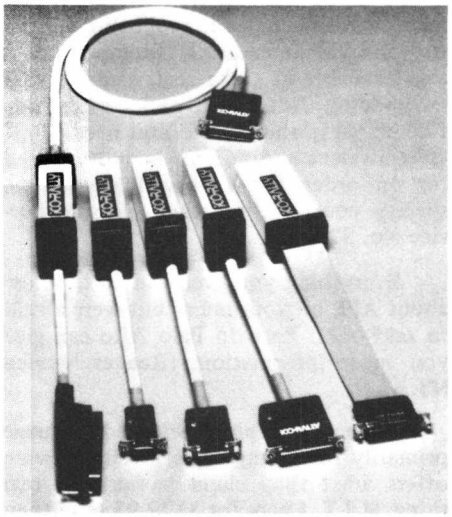
Your basic inexpensive tiny computer is the **Super Elf**, $106.95 for kit and $156.65 assembled (including high and low address options, speaker, transformer). Tiny BASIC for the 1802 processor, Super BASIC, S-100 expansion, color video, memory expansion to 64K, an editor and assembler, and suchlike are all options. (Reader Service **No. 343**)

The **Kaycomp II** comes in a solid-looking metal case. It has a nine-inch green display with 80 columns of 24 lines, eliminating, they say, the need for horizontal scrolling. Two 5¼-inch disk drives are included. The Z-80-based system software includes CP/M, a menu-driven operating system, a full-blown word processor, BASIC, Forth, an electronic spreadsheet, and a math program. All this costs $1795 (even the price is familiar). (Reader Service **No. 353**)

Hewlett-Packard has introduced a color pen-plotter, the HP 7470, compatible with most personal computers and many oscilloscopes and spectrum analyzers, and priced at a reasonable $1550. The 7470 uses a micro-grip drive system that obviates heavy moving arms and paper-transport drums and belts, eliminating some of the problems of conventional plotters. Two rotating "grit wheels" cause grit particles to penetrate the drawing medium at thousands of points, causing microscopic indentations indetectable to the naked eye. This machine weighs only 12.5 pounds! (Reader Service **No. 363**)

How would you like to instantly double the capacity of all your floppies, for only $9.39? The **Flip-Jig** is a gadget to mark and punch the jackets of diskettes so that you can use both sides. After all, both sides of a diskette are coated with magnetic material, but when the manufacturer puts it into its jacket, only one side is usable (we know there

The HP7470 color pen plotter (at left) is compatible with most personal computers.



The Match-Box from ICO-Rally (at right) provides a programmable pin and socket configuration for device compatibility.

are potential hazards, but if backups are used, the potential savings are at least as great. . .). (Reader Service **No. 373**)

Program it yourself on the **PROM-IT EDS** EPROM development system for the Apple II. Burn 8, 16, and 32K EPROMs, and down-load hex files from any computer or development system with an RS-232 port, for $129.50. (Reader Service **No. 383**)

Connect anything to anything with Ico-Rally Corp.'s **Match-Box**, a two-component computer cabling system with programmable pin and socket configuration for device compatibility. Users no longer need to unsolder leads and move pins. With Match-Box, they can neatly adjust the cables in a few minutes. Prices range from $19 to $53, depending on the number of conductors in the cable. (Reader Service **No. 393**)

Expanded disk storage for IBM personal computers provides up to 672K bytes in 5¼-inch floppy-disk drives. **Interface** prices range from $470 for one single-sided drive internal to the IBM cabinet, or $670 for one double-sided (336K) external drive with cabinet and power supply (Reader Service **No. 403**)

Plexa-Lok is a clear cover to protect your Apple keyboard. It locks into place, and can be used while the computer is running. Your $19.95 includes a 30-day money-back guarantee. (Reader Service **No. 413**)

. . . and Software Not Sold in Dress Shoppes

MicroMUMPS, ANSI MUMPS for most micros, is available from the University of California at Davis. Since the code is in the public domain, it costs only $33, including object code, applications, utilities and documentation. Source code is also available at extra cost, as are a year of updates. Additional packages include File Manager and Quest. You need CP/M or equivalent and 56K of memory. Write for an order form on which you can specify your configuration. (Reader Service **No. 379**)

Everything you ever wanted to know about APL on your micro but were afraid to ask? **APL Press** in Palo Alto can give you more information. (Reader Service **No. 423**)

Logo is a graphics-oriented language primarily for young users. Krell Software offers what they claim is the only real thing, **M.I.T. Logo**, for $179.95 (and they ask you not to buy anybody else's). Requires 64K (Reader Service **No. 433**)

Terrapin is an anybody else that offers a version of Logo called **Terrapin Logo Language**. (The language features a turtle.) This one costs $149.95 and is only for Apple II (also 64K). (Reader Service **No. 443**)

If you're going to sell software to the Department of Defense, you need Ada. Ada is primarily a process-control language, but it is also useful for business, finance, education, and mathematics, according to Supersoft, who sell **Supersoft Ada**, which they claim is the first true Ada compiler for microcomputers (everything for micros so far has been a subset). (Remember, this is a "true" compiler for a language for which the standard has not yet been validated.) It sells for $250, and requires 48K of memory on a Z80, 8080, or 8086/8088 system with CP/M. (Reader Service **No. 453**)

MicroPILOT is a command-oriented authoring language for educators. Teachers with no previous computer experience can create their own computer-assisted instruction materials. It costs $79.95 from Radio Shack, and requires a TRS-80 Model I or III with at least 32K of memory. (Reader Service **No. 463**)

A real native code hardware-independent BASIC compiler for CP/M? That's the claim for **BASIC/Z**. Input/output is supported at the source code level. All floating-point math is done in binary-coded decimal (BCD), to avoid round-off errors. Records can be blocked and deblocked. A SORT verb, part of the language, sorts 2000 elements in two seconds. Structured programming is supported with nested DO/UNTIL, WHILE/WEND, FOR/NEXT, IF/THEN/ELSE. User-defined functions are fully recursive, and can contain an unlimited number of statements. Error trapping routines can even handle BDOS errors, and PUSH/POP permits subroutine stack manipulation. BASIC/Z includes an interactive editor with global search and replace and complete syntax testing during entry. A debugger offers line tracing, error line retention, and "stepping" one line at a time through a compiled program while continuously displaying the values of selected variables. Sounds good. BASIC/Z runs on CP/M systems with addressable cursor, and costs $345. (Reader Service **No. 473**)

SMAL/80 (Structured Macro Assembly Language) "combines the logical power, versatility, and convenience of a compiled, structured high-level language like Pascal, Ada, or C with the efficiency and flexibility of an assembly language." In addition to assembly language statements, SMAL/80 uses constructs like BEGIN . . . END, IF . . . THEN . . . ELSE . . ., and LOOP . . . REPEAT, and is not field-fixed, permitting indented statements for easier reading. It also uses symbolic notation to reflect what the processor is actually doing, such as A = B instead of MOV A,B and A = NOT A instead of CMA. It's a compiler, not an assembler, an extension by Dr. Charles Popper of SMAL, the language he developed in 1974 at Bell Telephone Labs. The package includes compiler, linker, macro processor, and macros that translate 8080 into SMAL/80. Chromod Associates sell SMAL/80 on CP/M disks for $150. (Reader Service **No. 483**)

Printers Not Found in Printshops

How about a letter-quality daisy wheel printer for $1195? The **DTC 381** merges new technology with a refurbished Diablo HiType I print mechanism. Serial RS-232 interface, baud rates from 50 to 19,200, plastic printwheels, proportional and incremental printing, bidirectional printing, optional Centronics-compatibility, a mobile wheeled stand with paper tray and side shelves, etc., etc. (Reader Service **No. 493**)

The **Transtar 140** Diablo-compatible 40-cps serial daisy wheel printer lists at $1695. (Reader Service **No. 309**)

If you have an IBM 50, 60, or 75 electronic typewriter, you can get a "smart" parallel printer interface for $499.95, or serial for $649.95. The **Tlec Mark II** or **III** features software independence, and text formatting capabilities, such as proportional spacing, right justification, mixed pitches, page titling and numbering, hanging indents, centering, underscoring, intelligent margins and tabs, boldface, etc., and 17 cps throughput. (Reader Service **No. 319**)

MicroFazer is a 64K printer buffer for all popular micros and parallel printers. It receives and stores up to 64K of data at 4000 cps, and then transfers the data to the printer as rapidly as the printer can handle it, meanwhile freeing the computer for other processing. Prices range from $159 for an 8K buffer to $299 for 64K. (Reader Service **No. 329**)

(Continued on page 25)

PDP-1802

A Hardware Simulator in Assembly Language

The program presented in this article permits the execution on COSMAC 1802 systems of programs written in the machine language of the Digital Equipment Corporation PDP-8 minicomputer. The article is in two parts; part 1 describes the PDP-8 instruction set and front panel functions, as well as an octal console monitor program for the 1802 which allows loading, examining and changing 12-bit memory locations using octal notation. Part 2, to be published soon in *DDJ*, presents the PDP-8 instruction interpreter for the 1802, and the I/O and interrupt facility.

The original PDP-8 was the first minicomputer to be widely adopted, and in many respects it heralded the minicomputer revolution; it was the 8008 of minis, and brought computer power to a far broader spectrum of users than had previously been the case. Partly as a result, an immense amount of free or low-cost software exists for the PDP-8, much of which can be executed using the emulator described here. Many of the software and systems concepts first developed for the PDP-8 have since gone on to find application elsewhere in the DEC line, and by extension in the computer field at large.

The first PDP-8 was made up of discrete devices, with a massive power supply and fan system to match. Later models used integrated circuits, and both size and power requirements were greatly reduced. The emulator described here, which I call the PDP-1802, is based on the first PDP-8 (the "straight eight"), with certain modifications closer in operation to the behavior of the PDP-8/I and PDP-8/L, which were later models. These modifications are described in part 2.

The emulator maps a virtual 12-bit word address space onto the 8-bit word space of the microcomputer system, and performs the necessary address conversion and data packing and unpacking. Thus 4096_{10} twelve-bit words (a PDP-8 "field") map into 6K of eight-bit bytes. Since the emulator requires only about 1½K bytes, including the octal console monitor, a PDP-8 with a single field can be emulated on an 1802 with fewer than 8K bytes. Of course, it is possible to write and run small PDP-8 programs on 1802 systems with as few as 2K bytes.

Hardware requirements are minimal. Most 1802 systems use either a toggle-switch front panel, or (more commonly) a hexadecimal keypad. In either case, this is the minimum needed to enter and run programs. Certain PDP-8 switch-register functions (described below) cannot be supported on systems using only a software-scanned hex keyboard such as that on the RCA VIP; such users will need to provide software or hardware alternatives.

Christopher L. Hamlin

Copyright © 1982 by Christopher Hamlin. All Rights Reserved.
2202 Hindle Lane, Bowie, MD 20716.

The PDP-8

The front panel of the PDP-8 has twelve toggle switches for setting addresses or data into the switch register. These permit one to address 4096_{10} twelve-bit words directly. Three more switches allow manual selection of any of eight such 4096_{10} word fields. The PDP-1802 emulator assumes there is one field whose first address is 0000_8, though it could readily support multiple field operation.

The remaining switches are:

Load Address: Causes the address set in the twelve switches to be placed in the address register.

Deposit: Causes the word set in the twelve switches to be placed in the memory location pointed to by the address register. The address register is then incremented, allowing loading of successive memory locations from the panel switches.

Examine: Causes the value in the memory location pointed to by the address register to be placed in the accumulator and displayed on the front panel, in the twelve lamps showing the accumulator content.

Start: Causes execution to begin at the address in the address register (usually placed there by using the Load Address switch). Various registers and flags are initialized in so doing.

Continue: Same as Start, but without the initialization step.

Single Cycle: When on, this switch permits stepping through a program machine-cycle by machine-cycle by repeatedly pressing the Continue switch.

Single Instruction: Same as Single Cycle, but executes a complete instruction.

In addition to the accumulator, the states of the memory buffer, program counter, and memory address register are displayed by sets of twelve lamps, and one lamp shows the state of the link bit (similar to a carry bit for the 1802 DF).

For PDP-8 models equipped with hardware-extended arithmetic elements, an additional set of lamps shows the state of the multiply register (not included in the PDP-1802 emulator).

Finally, a set of lamps shows the current machine cycle, and the type of instruction or machine state.

PDP-8 Instructions

The PDP-8 instructions are simple and are easy to learn quickly.

All instructions are twelve bits long. The left-most (most significant) three bits (called bits 0, 1, and 2) form the operation code; there are eight of these. The instructions with op codes 0_8 through 5_8 treat the remaining nine bits identically, and are called memory reference instructions. The memory reference instruction op codes and their mnemonics are given below.

Here is how the low nine bits of the memory reference instructions are treated:

The lowest seven bits form the low part of the address with which the instruction is concerned, and can address any word on one page (a page is 128_{10} twelve-bit words).

Bit 4 (fifth from the left) controls where the other five bits of the full address with which the instruction is concerned will come from; in other words, bit 4 determines which page contains the address the instruction refers to. If bit 4 is zero, then the address is a location in page 0, specified by the low seven bits. If bit 4 is one, then the address is on the current page (the page on which the instruction itself resides).

Bit 3 (fourth from the left) controls direct and indirect addressing. If it is zero, direct addressing is used, and the instruction is carried out using the location specified by bits 4 through 11 as the effective address. If bit 3 is one, indirect addressing is used. The effective address for the instruction is that pointed to by the content of the location specified by bits 4 through 11. Indirect addressing permits operations to be carried out over the whole field addressed by twelve bits.

To summarize, memory-reference instructions operate on locations selected directly or indirectly through page 0 or

the current page.

The actions of the memory reference instructions are as follows:

(0) – AND: perform logical AND between the content of the location specified and the content of the accumulator, leaving the result in the accumulator. The memory location's content is unchanged.

(1) – TAD: perform two's complement addition between the specified location's content and the accumulator content; the result is left in the accumulator. Any carry complements the link bit. In two's complement notation, the values 0000_8 through 3777_8 are positive; 4000_8 through 7777_8 are negative, with 7777_8 representing -1.

(2) – ISZ: increment the content of the specified location; if the result is zero, skip the next instruction (that is, the instruction following the ISZ).

(3) – DCA: deposit the content of the accumulator in the specified location, then clear the accumulator by placing 0000_8 in it.

(4) – JMS: jump to a subroutine. The address of the location following the JMS instruction is automatically placed in the specified location, and execution proceeds with the instruction immediately following the specified location. (A return is thus simply a jump indirect to the first word of the subroutine).

(5) – JMP: jump; transfer execution to the location specified.

The remaining op codes are 6 and 7.

Op code 6 is reserved for input and output instructions. Since these are hardware dependent, only the I/O instructions governing operation of the teletype and paper tape reader/punch are implemented in the PDP-1802 emulator. They are described in the next part of this article.

Op code 7 covers operations on the accumulator, link, and switch register, as well as conditional jumps. It also governs operation of the extended arithmetic element if this is used; EIA instructions are not implemented in this emulator.

The op code 7 instructions are called the "operate microinstructions." There are two groups of operate microinstructions, called Group 1 and Group 2. Bit 3 governs the choice of group; if bit 3 is 0, the Group 1 operate microinstructions are selected. If bit 3 is 1, the Group 2 operate microinstructions are selected so long as bit 11 (the rightmost) is a 0 (if it is a 1, then the extended arithmetic element is invoked).

The function(s) to be performed by the instruction are determined by setting one or more bits to 1. The bit assignments for Group 1 are as follows:

Bit 4: Clear the accumulator (set it to 0000_8);
Bit 5: Clear the link (set it to 0);
Bit 6: Complement the accumulator (change 0's to 1's and vice versa);
Bit 7: Complement the link;
Bit 8: Rotate the accumulator right (through the link);
Bit 9: Rotate the accumulator left (through the link);
Bit 10: Perform a rotate operation selected by bit 8 or 9 once if 0, or twice if 1;
Bit 11: Increment the accumulator.

More than one of the function selection bits may be set to 1 in a single operate instruction; that is why these are called microinstructions. By setting bit 4 and bit 6 in a Group 1 microinstruction, for example, the accumulator would be cleared and complemented, leaving it filled with 1's. Any logically admissable combination of microinstructions may be used (it would not, for instance, be logical to specify both a rotate left and a rotate right). The precise sequence in which combined microinstructions are executed is described in part 2 of this article.

The Group 2 operate microinstructions include conditional skips based upon the state of the accumulator and link. These conditional skips work in an ingenious way which requires special discussion. The other Group 2 microinstructions are therefore presented first:

Bit 4: Clear the accumulator (same as Group 1);
Bit 9: Perform logical OR between the switch register and the accumulator contents, leaving result in the accumulator;
Bit 10: Halt; stops processor operation.

The conditional skip microinstructions test the accumulator or the link or both. If the specified condition is met, the instruction following the operate microinstruction is skipped. There are two distinct sets of tests which are performed to determine whether a skip will occur. Bit 8 (called the reverse sensing bit) determines which set of tests is to be performed. These two sets of tests are treated separately:

Bit 8 is 0

Bit 5 set to 1: perform skip if accumulator is minus;
Bit 6 set to 1: perform skip if accumulator is zero;
Bit 7 set to 1: perform skip if link is non-zero.

If more than one condition is being tested (i.e., more than one bit is set), only one of the tests called for need be true for the skip to occur. In other words, the results of the tests called for are logically ORed together, and if the result is 1 (true) the skip takes place.

Bit 8 is 1

Bit 5 set to 1: perform skip if accumulator is positive (zero is positive);
Bit 6 set to 1: perform skip if accumulator is non-zero;
Bit 7 set to 1: perform skip if link is zero.

Notice that the conditions are reversed if bit 8 is 1. In addition, when the reverse sensing bit (bit 8) is set, the results of the conditions tested are logically ANDed together, instead of being ORed. Thus *all* conditions for which tests are called must be met for the skip to occur. For example, if bit 8 is 1, and bits 5 and 7 are set to 1, the skip will occur if and only if the accumulator is positive (which includes zero) and the link is zero.

The final Group 2 operate microinstruction is a special case: if bit 8 is 1 and bits 5, 6 and 7 are 0, an unconditional skip is forced.

The manner of combining Group 2 conditional skip microinstructions makes it possible to perform in a single instruction many kinds of tests which are important in efficient logical and arithmetic data processing.

The final PDP-8 op code, 6, will be treated in part 2 along with the discussion of PDP-8 interrupt servicing.

The Autoincrement Registers

Many computations involve the need to step through long tables of data one word at a time. In order to simplify (and speed up) operations of this kind, the

PDP-8 architecture provides a set of autoincrement registers. These aren't really separate registers; they are simply memory locations 0010_8 through 0017_8 inclusive. Here is how the autoincrement operation works:

If, and only if, one of the locations 0010_8 through 0017_8 is addressed *indirectly* by a memory reference instruction, then the content of that location is incremented by one *before* it is used to form the effective address (of the operand) of the instruction. The incremented content remains in the location; the content will be incremented again the next time the location is addressed indirectly. This is known as autoindexing. The technique allows for rapid stepping through tables without having to use ISZ instructions to update the address (notice that no separate instruction is needed to implement this function). Bear in mind that you must be careful not to address one of these locations indirectly if you do not wish its contents to be incremented. Autoindexing is fully supported on the PDP-1802.

The Octal Console Monitor

The front panel functions of the PDP-8 are emulated by a monitor using an ASCII keyboard and CRT display. (Provisions for substituting a hex keyboard and LED display are described too). On entry to the emulator, the monitor issues a carriage return/line feed and awaits imput. At this point, only two types of input are recognized: a "C", or an octal digit. If a "C" (for Continue) is typed, execution of an already entered PDP-8 program continues with values which have been set up in the registers and buffers. This can be used to resume execution after a Halt, for instance. If anything but a "C" is entered, the monitor will take the first four octal digits entered (ignoring any other characters) as an address. The content at that address is immediately displayed in octal digits, and the cursor halts on the same line. At this point the monitor will accept only a space, an octal digit, "C", "S", or "L" as input. Pressing the space bar causes successive locations and their content to be displayed. If an octal digit is entered instead, then the first four octal digits entered will replace the old content at that location. Entering "L" (for Load) causes issuance of a CR/LF; the monitor then waits for four octal digits, forming a new (virtual) memory address to be examined or modified. "S" causes execution to start from the location currently displayed by the monitor, after initialization of the PDP-8 registers. "C" causes execution to continue from the currently displayed address without initializing the registers.

Other Panel Functions

The PDP-8 single-step function and switch register are implemented in the PDP-1802 by closely related methods, and so are described together. It is assumed that the 1802 system has either a hex keypad or a set of eight toggle switches for entering data and addresses, and that an additional switch connected to one of the 1802 EF lines is used to signal the presence of data in the switches or keypad latch.

Closing the switch on the EF line causes execution to be suspended upon completion of the current PDP-8 instruction. The content of the program counter, the memory address buffer, and the accumulator are shown on the CRT screen in octal. After the EF line switch is released (opened), the emulator is ready to set up the pseudo switch register. The high four bits of the twelve-bit switch register content are placed in the low half of the hex keypad latch or panel switches. The most significant bit of this byte will determine whether single-stepping mode is to be entered. If that bit (which is not one of the twelve-switch register bits) is a 1, a single instruction will be executed. If it is a 0, normal execution is enabled.

Pressing (closing) the EF line switch and holding it pressed loads the byte, setting the most significant four bits of the pseudo switch register and also setting or resetting the single-step flag. While the EF line switch is still held closed, the low eight bits of the pseudo switch register are now set up on the keypad or panel toggles. When the EF line switch is released, the pseudo switch register will have been set up, and single stepping will have been enabled or disabled. Execution will continue accordingly.

Notice that this procedure, although it permits carrying out any operation of which the PDP-8 switch register functions would be capable, does not work in the same way it does on the actual PDP-8. In effect, the EF line switch results in actions like those obtained by successively pressing Halt and Continue on the PDP-8 (when the emulator's single-step bit is 0) or like the single instruction switch (when the single-step flag bit is 1). The major difference is that computer operation is halted whenever the switch register setting is changed.

It is important to observe that the procedure just described only sets up the pseudo switch register. In order to make use of its content in a program, the normal PDP-8 microinstruction for ORing the switch register into the accumulator must be used. The emulator does not allow for the single cycle function, because the structure of operations by which instructions are executed does not resemble the states of the PDP-8 cycles.

It should now be possible for readers to experiment with writing and entering PDP-8 programs, using the octal monitor. In the concluding part of this article, the input and output functions for the teletype and paper tape reader/punch are described, along with the procedure for emulating interrupts. The operation of a software UART (universal asynchronous receiver/transmitter) for the teletype, included as part of the emulator, is also explained. The implementation of the PDP-8 instruction interpreter is outlined, and considerations affecting adaptation of the emulator to different hardware configurations, as well as relocation of the program (perhaps for installation in an EPROM) are discussed. ▶▶J

Homebrew Lives:
Multi-68000s in a Personal System

The computer system described in the following paragraphs is an attempt on the part of myself, Darryl Uchitil, and Jim Hannas to design and build a home computer at the current state of the art in microprocessor-based computers. It has a simple bus design, supports a very sophisticated color raster display, a number of different types of mass-storage devices, and a sophisticated multi-tasking operating system.

The first design session for the graphics display began in the early spring of 1979. I was, at that time, working for the University of Wisconsin at Milwaukee as the graphics consultant for the university computer center. In addition to the usual collection of Tektronix graphics devices, the computer center had a DEC GT-46 vector interactive graphics system. The GT-46 is a PDP-11/34 computer with a 16"vector display capable of drawing vectors with 10 bits of X and Y resolution. If you have ever seen a vector display rotating an image in real-time you would know why Jim, Darryl and I all sensed a need for such a device on our own computers.

The initial design meetings we had centered on the idea of getting a vector display running on the S-100 systems we had at home, with at least the resolution of the GT-46. A prototype for the 1024 x 1024 vector display was finished in late spring by Jim but presented all of us with a problem. While the vector display did work, it was limited in the number of lines that it could draw before the entire screen began to flicker. There was no way to speed up the drawing rate of the vectors without going to an analog vector generator, something which we really didn't want to do. In addition, there was no practical way to ever build a high-resolution vector display capable of displaying color. As a result, it was decided that the graphics display would have to be a raster rather than a vector display system.

The decision to abandon the design of a vector graphics display in favor of a raster system represented a real problem in generating images via the computer. A vector display requires only the end points of a line segment to be computed to draw a line, whereas every point within a line must be computed and drawn a dot at a time on a raster display. There was no way a Z-80 would ever be able to perform the calculations fast enough for updating a raster display. Since a Z-80 wouldn't be fast enough, we would have to build faster computers.

The processor we decided upon at that time for the graphics computer was the Motorola 68000. Although very little information was available on it in the summer of 1979, we felt that the 24-bit address space and 32-bit registers would be very useful for manipulating all of the memory required for a high-resolution color display.

The idea of multiple 68000s within the computer evolved because of our desire to maximize performance of the graphics display while also being able to support a multi-tasking operating system. I was able to get enough information on the 68000 processor by early fall to begin the slow and painful process of bootstrapping the software needed to get the processor going. A FORTRAN cross-assembler was a byproduct of the initial work done to write a small debugger for the 68000.

At the same time, work had begun on defining the hardware required to implement the computer. The first decision was the system bus. We all felt the S-100 was not an acceptable bus for a multiple processor system. The final size and configuration of the boards used ended up to be a somewhat arbitrary decision based on the availability of a large number of wire-wrap boards I was able to locate from a bankrupt electronics company. Since we knew that a "de facto" standard for 68000s was still a long way away, there was little we could lose by not using a defined bus.

We have called the bus which we are using for our computers the PUNIBUS. The signal definitions for the bus are shown in Figure One. The PUNIBUS is in fact two distinct busses. The first bus is used between 68000 processors and devices such as RAM and the graphics control board which the 68000 processor board can access. The second bus is a high-speed, 16-bit bi-directional data path between processors and peripherals. The details of the peripheral bus (PBUS) appear in a later section.

The timing specifications for the PUNIBUS are similar to those of the M68000 microprocessor. Most of the signals on the bus are buffered versions of the signals coming from the processor or memory management unit. There are a few signals which were left off the bus that are available for the control of 68000 peripherals, and a few special signals were added that provide byte-wide parity and multi-level, daisy-chained interrupts.

We are currently using dual-width (2 connectors) and a triple-width (3 connectors) cards in implementing the system. Dual-width cards are used wherever no PBUS communications are needed, or where the PBUS connector is not used as an inter-card jumper between PC boards.

Computer Hardware Implementation

The 68000 multiprocessor system is made up of two 68000 processor cards, a two-card raster graphic display controller, a system memory board capable of holding up to one megabyte of memory, and a collection of special-purpose intelligent I/O boards for operation of serial interfaces, cartridge and floppy disks, and 9-track magnetic tape. The interconnection of the boards in the system is shown in Figure Two.

M68000 Processor Card

Each processor card has an 8MHz 68000 processor, 68451 memory management unit, an interrupt generator which provides a programmable interval clock used by the operating system as the event sequencer, and 8 sockets for a rudimentary EPROM monitor and bootstrap program. In addition, one processor has a CMOS time-of-year clock used to maintain the system's time and date.

Raster Graphics Controller

The graphics display in the multiprocessor is a device capable of displaying 256 unique colors from combinations of 256 shades of red, green, and blue upon a 640 x 480 pixel color video display. A block diagram of the graphics controller is shown in Figure Three. The image as displayed on the screen is stored in the display memory as a series of 8-bit pixels. Display memory is fetched eight pixels (64 bits) at a time to increase the

by Al Kossow

Al Kossow, Medical College of Wisconsin, Department of Physiology, 8701 Watertown Plank Road, Milwaukee, WI 53226.

apparent access speed of the dynamic memories and to allow the 68000 processor to interleave memory accesses. These eight-bit values are used as the adress of a 1K by 24-bit high-speed color mapping memory. The output of the color map is used as the 8-bit input to three Intech high-speed video DACS which output to a 19" RGB color monitor. Additional features of the graphics controller are the ability to pan and zoom within the display memory.

Dynamic Memory Cards

The original memory board we developed was a dual-width card which used 16K dynamic RAMs and was capable of a maximum of 128K bytes of RAM per card. Because of the rapid decrease in price of 64K RAM devices, and because of the need for a massive amount of memory for the graphics display memory, we have switched to a triple-width card capable of holding twice as many chips per board. Our graphics display memory boards and the system memory boards are built on the same printed circuit board which provides a bussed memory array with gridded power and common lines on the top of the board, and a wrapped-wire prototype area on the lower half of the board. Doing the memory boards this way provided a well-bussed memory array which is essential for correct operation of dynamic memories, while also allowing us the ability to prototype the refresh and control logic.

Intelligent I/O Control and the PBUS

One of the biggest problems with a multiple CPU computer is the allocation and sharing of resources between CPUs. Normally, peripherals are connected to one processor, with other processors accessing the peripherals via shared memory or a special dual-porting arrangement on the peripheral. We have chosen to consider commonly used peripherals (such as mass storage devices and terminal lines) as being universally sharable resources and have connected those commonly used devices on a special high-speed parallel peripheral controller bus called the PBUS. CPUs within the system have th ability to request peripheral controllers to perform operations via PBUS "channel" command packets. In addition, processors can issue requests to other processors to allow inter-processor data transfers.

Figure One

PUNIBUS Signal Definition

	Pin	Name	Pin	Name
	A	Common	1	Common
	B	DATA0	2	DATA1
C	C	DATA2	3	DATA3
o	D	DATA4	4	DATA5
n	E	DATA6	5	DATA7
n	F	DATA8	6	DATA9
e	H	DATA10	7	DATA11
c	J	DATA12	8	DATA13
t	K	DATA14	9	DATA15
o	L	PARITY1	10	PARITY2
r	M	+5V	11	+5V
	N	Upper DATA Strobe	12	Lower DATA Strobe
A	P	DATA Transfer ACK	13	Bus Error
	R	VALID Peripheral ADR	14	Write
	S	+12V	15	+12V
	T	INT ACK Out	16	–
	U	INT ACK In	17	Non Maskable INT
	V	-12V	18	-12V
	W	Interrupt1	19	Interrupt2
	X	Interrupt3	20	Interrupt4
	Y	Interrupt5	21	Interrupt6
	Z	Common	22	Common

	Pin	Name	Pin	Name
	A	Common	1	Common
	B	ADR1	2	ADR2
C	C	ADR3	3	ADR4
o	D	ADR5	4	ADR6
n	E	ADR7	5	ADR8
n	F	ADR9	6	ADR10
e	H	ADR11	7	ADR12
c	J	ADR13	8	ADR14
t	K	ADR15	9	ADR16
o	L	ADR17	10	ADR18
r	M	+5V	11	+5V
	N	ADR19	12	ADR20
B	P	ADR21	13	ADR22
	R	ADR23	14	ADR Strobe
	S	+5V Battery	15	+5V Battery
	T	Bus Grant Out	16	Bus ACK
	U	Bus Grant In	17	Bus Request
	V	-5V	18	-5V
	W	IOSEL	19	Memory Disable
	X	–	20	Reset
	Y	–	21	Clock
	Z	Common	22	Common

(Figures 2 and 3 on next page)

There are a few devices (such as the graphics controller) which require such a high amount of processor interaction that they are connected directly to a M68000 processor via the PUNIBUS. Currently, the graphics display is the only device which requires such tight coupling to a processor.

An intelligent I/O card normally consists of a PBUS interface, the interface hardware required of the peripheral(s) being controlled by that board, and either a Z-80 or a bipolar bit-slice microprocessor. Currently, all I/O cards are using Z-80s as the control computer but we suspect that we will need the extra speed of the bit-slice devices for some of the fancier interfaces we would like to do (such as the network link).

System Software

The major component of the software for the multiprocessor is the BERT multi-tasking operating system. BERT is based on ideas from UNIX, VAX/VMS and RSX-11M. The majority of the system is being written in C. The operating system is currently being bootstrapped via a C cross-compiler developed on the RSX system at work. Once the opsys kernel, the file system, and the command processor are running on the multiprocessor, the C cross-compiler, assembler, and linker (which were all written in C) will be cross-compiled, linked and moved to the 68000 where the balance of the development will be done.

Software for the graphics hardware will be developed as a parallel effort with the operating system, by adapting CLICS (C Language Implementation of the Core Standard, which was developed by the Department of Defense) to the graphics hardware on the multiprocessor.

The FORTRAN Cross-Assembler

When this project was first started, I was totally ignorant of UNIX or the C language. Since that time, I have become hooked on the use of C for systems programming and operating systems development. Prior to this change, however, I wrote a cross-assembler for the 68000 which was writen in FORTRAN to run on either RT-11 or RSX-11. It is of little use to me now, and I thought that someone else might have use for such a program. The *DDJ* editor tells me he will publish the listing (it is quite long) only

Figure Two

M68000 Multiprocessor Configuration

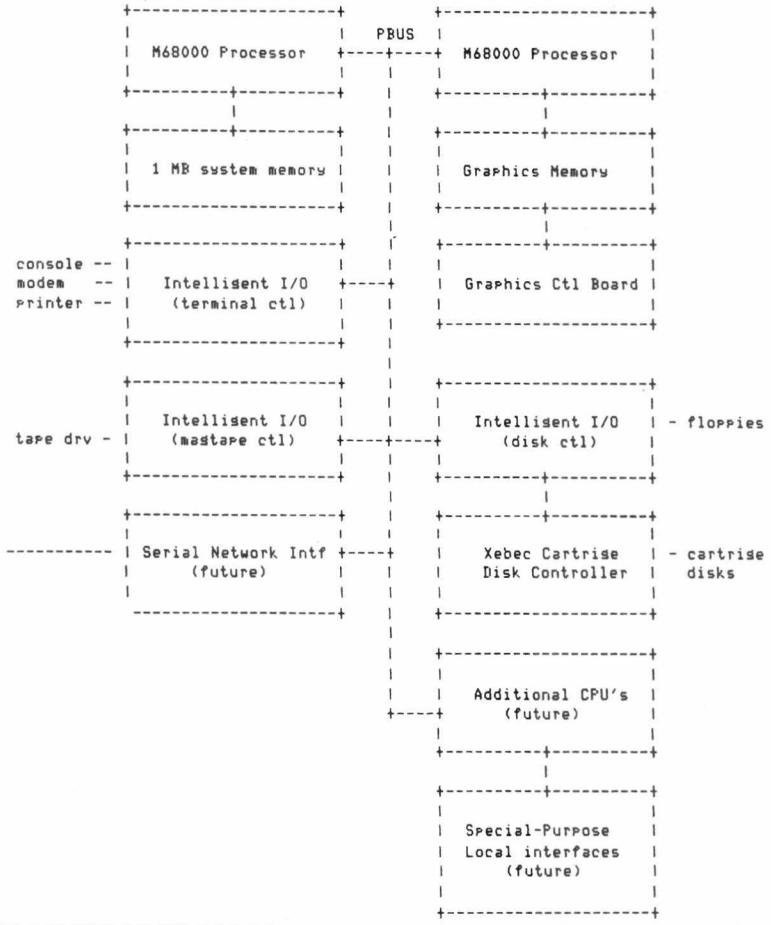
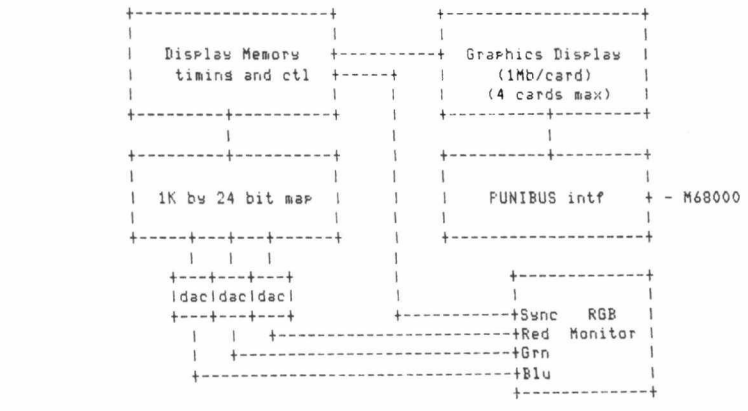


Figure Three

Raster Display Block Diagram



if a considerable number of readers clamor for it. The program currently requires a few assembly-language subroutines to perform 32-bit arithmetic and binary operations. They would have to be rewritten to be of use on a microsystem. I also do not know if the program will compile using Microsoft FORTRAN. Since I do not have the compiler, I have no way of knowing if the program will compile or not.

DDJ

Of Interest

(Continued from page 13)

Operating Systems Are Not Phone Companies

CP/M-86, for 8086 systems, from Digital Research is completely compatible with all Z-80 and 8080 CP/M programs, in addition to its own 8086 8-bit cross-assembler and 16-bit assembler, DDT-86 debugging tool and assembler/disassembler, and availability of compilers that need more than 64K to run efficiently (FORTRAN 77, etc.). The **8086 Dual Processor Option (DPO)** is a single board from Veritas that adds CP/M-86 to your IBM Displaywriter or TRS-80 Mod II. It costs $1500, including CP/M-86. (Reader Service **No. 339**)

MicroShell from New Generation Systems is a Unix-like operating system that relocates itself under CP/M and remains compatible with CP/M. MicroShell automatically searches multiple user-specified disks for programs and files, redirects or appends output to a file with optional screen echo, sends console output of one program to console input of the next ("pipes"), supports multiple commands on one line, directly reads and executes commands from a file without the necessity of Submit processing, retains all CP/M CCP functions, has user-definable prompts, and can be customized to the user's environment. MicroShell is available on disk for 8080/8085/Z-80 CP/M systems at a cost of $150. (Reader Service **No. 349**)

Computer Coverage Is More Than Dust Covers

Protect your equipment with personal computer insurance coverage from (who else?) **Personal Computer Insurance**. No-deductible insurance that covers everything *except* data costs $50 a year, with extended business coverage for $60. (Reader Service **No. 359**)

All-risk computer insurance coverage for businesses that *includes* data is offered by Kemper. They call it **BEEP** (Business Electronic Equipment Policy) and offer protection with various deductibles from $100 up. Each policy is different, so you'll have to call for prices. You might also ask them about home computer coverage, particularly if you're self-employed. (Reader Service **No. 369**)

DDJ

An Improved LET Processor for LLL Basic

by Norman E. Thagard and George Glenn

Norman E. Thagard, M.D., 528 E. Castleharbour Drive, Friendswood, TX 77546.

Norman Thagard is a mission specialist astronaut with NASA. He writes, "Ownership of personal computers among employees at Johnson Space Center is common. Given the preponderance of scientists and engineers here, that is expected.... There are a number of desktop machines at the center devoted to system simulations..."

Of himself, Thagard says, "Currently, I am the astronaut office point-of-contact (along with female astronaut Shannon Lucid) for STS-7, which is scheduled for launch next April. Unfortunately, that does not mean that I am flying on the mission but I am learning a lot about proximity operations. This is because this is the first shuttle misssion where we will deploy a payload, fly around it, then pick it back up."

George Glenn is an engineer under contract to NASA who is currently working on the software for the Remote Manipulator System which flies on the space shuttle.

Lawrence Livermore Lab Basic has a restricted assignment (LET) statement. LLL Basic expressions can be no more complex than

<LLL Basic Expression> →
operand operator operand

where *operand* may be a constant or a variable, and operator is one of the arithmetic operators *, /, +, or -. Borrowing from compiler theory, we overcame these limitations with a patch routine which incorporates certain principles of a form of shift-reduce parser known as operator-precedence.

Because LLL Basic is an interpreter rather than a compiler, the patch is written a bit differently than textbook examples but the idea is similar. A parser recognizes sentences in a language and this is true for natural languages as for programming languages. In the Basic language sentences are statements of which LET statements are particular examples. Our patch, then is merely a recognizer for LLL Basic LET statement right sides. LET statement right sides are merely arithmetic expression which satisfy the following rules (called productions):

```
Expression → Expression + Term
               or
             Expression - Term
               or
             Term

where
     Term → Term * Factor
               or
             Term / Factor
               or
             Factor

and
   Factor → Variable
               or
             Constant
               or
             Expression
               or
             GET(LLL Basic
                 Expression)
```

The "→" means the right side is an example of the left side. Variable and Constant are the allowable LLL Basic variables and constants.

It follows from the above rules that complex right sides are permitted and are evaluated as dictated by parentheses if present, according to the operator precedence *, /, +, - if no parentheses are present, and left-associatively if operators are of equal precedence. This means that if A=1, B=2, and C=3, then (A+B)*C=9, and A+B*C=7. Arithmetic operators are associative, i.e., (A+B)+C=A+(B+C). However, the patch would evaluate A+B+C as though written (A+B)+C (left-associatively).

A shift-reduce parser takes the input one item (usually called a token) at a time and either shifts (pushes) that item to a stack, reduces (pops) an item(s) from the stack, or flags a syntactical error. It terminates its operation when end-of-line is detected. The decision about which action should occur is based on the results of a comparison between the relative precedence of the item at the top of the stack and the item just read. In theory, only one stack need be maintained. However, since LLL Basic is an interactive interpreter, it was easier to determine operand *values* and store them on a separate (operand) stack to implement this patch. The shift-reduce decisions are based on the operators and therefore only the operator stack is involved in such decisions so the operand stack does not complicate the operation of the patch.

To briefly explain how an operator-precedence parser works, we can take the expression A+B*C used earlier. Operand A is read and its value determined and stored. Often, a token is chosen to represent A. This token, really just a placeholder, is then pushed onto the stack on top of another token used to mark the bottom of the stack. As mentioned before, this is not done in this patch. Rather, the value of A is pushed onto a separate operand stack. Next, the operator "+" is read and, as it has higher precedence than the bottom of stack marker, it is pushed onto the operand stack immediately. Next, B is read and its value is placed onto the operand stack above the value for A already there. The "*" operator is read and its precedence compared to that of the "+" at the top of the operator stack. Since "*" has higher precedence than "+", we know that the operation A+B should not be performed yet so "*" is pushed on top of the "+". Now C is read and its value pushed onto the operand stack on top of A and B. Our two stacks now appear as

operand stack	operator stack
C	*
B	+
A	$
	(bottom of stack)

The end-of-line marker is read next. This forces the parser to pop one operator from the operator stack and two operands from the operand stack to perform the operation B*C, placing the result back on the operand stack. The stacks now appear as

operand stack	operator stack
(result of B*C)	+
A	$

The end-of-line marker forces us to continue to pop the stacks until empty, so now the "+" is popped along with the result of B*C, and A. The indicated operation is carried out to yield A+(B*C) which is the expected result.

The left parenthesis, if it appears in the input expression, is automatically pushed onto the operator stack but is assigned the lowest precedence so that the next operator will always be pushed on top of a left parenthesis. On the other hand, when popping operators off the stack, the left parenthesis signals that the necessary reductions have been carried out and reading the input can resume. In compiler lingo, the left parenthesis is said to define the left side of a "handle." In this case, the handle is of the form (E) where E denotes an arbitrary (arithmetic) expression. The fundamental operation of the recognizer is identifying such "handles" in the input and replacing them by their value. Whatever the form of E in (E), upon encountering a right parenthesis, the stack will be popped, the appropriate operations performed, and the process repeated until the matching left parenthesis is popped. If no left parenthesis is found an error message is generated. The recognizer permits the use of a unary minus. As with the left parenthesis, a unary minus, when read, is automatically placed on the operator stack. In contrast to the left parenthesis, however, the unary minus is assigned the highest precedence of all. As a result, an expression such as A+-B+C is evaluated as though written (A+(-B))+C. The expression A+-(B+C) is evaluated as A+(-(B+C)) because of the rule that the left parenthesis is automatically pushed onto the operator stack.

To use the new expression processor, called EXEVAL, requires a patch in the LET processor of LLL Basic. The patch involves the six consecutive bytes at 016117-016124 octal. In the first three bytes we "no op" the call to ICP8. In the second three bytes we call EXEVAL rather than EVAL. We have located EXEVAL at 020004Q but it is likely that there is a great deal of variation among LLL Basic memory maps from user to user. So, if you need to relocate the patch the only critical feature is that the least significant address byte of table VALTAB must be 000Q. Shown below is the required change to the LET processor.

Change:

```
016117 315 373 025  CALL ICP8
016122 315 335 031  CALL EVAL
```

To:

```
016117 000 000 000  NOP NOP NOP
016122 315 004 040  CALL EXEVAL
```

(Continued on page 28)

LET Processor

Listing 1

```
FUNCTION Expression_Evaluator (EXPRESSION_VALUE)
    PERFORM Initialization
    DO UNTIL End of source line is reached or a syntax error is detected
        PERFORM Get_Next_Operand_Value
        IF No syntax error was detected THEN
            Store the operand value obtained on the operand stack and
            increment the operand stack pointer
            DO UNTIL End of source line is reached, a syntax error is de-
                     tected, or an arithmetic operator is encountered
                PERFORM Get_Next_Operator
    IF A syntax error has been detected THEN
        Setup syntax error annunciation
        ERROR EXIT
    ELSE PERFORM Cleanup_Processing
        EXIT
END

PROCEDURE Initialization
    Set the left parenthesis counter to zero
    Set the operand stack pointer to the first position of the operand stack
    Set the operator stack pointer to the first position of the operator stack
    Store the empty stack identifier on the operator stack
END

PROCEDURE Get_Next_Operand_Value
    CALL ICP8 to get the next character of the source line; syntax error is
         indicated if end of line is reached
    IF The end of line was not reached THEN
        IF The character is a left parenthesis THEN
            PERFORM Process_Left_Parenthesis
            ELSE IF The character is a minus sign THEN
                PERFORM Process_Unary_Minus
        ELSE CALL VALUE to get the operand value; syntax error is
                  indicated if the line item is not an operand
END

PROCEDURE Get_Next_Operator
    IF More characters remain in the source line THEN
```

(Continued on next page)

Note that this patch affects only the right side of LET statements. Other calls to the EVAL subroutine are left in place. This means, for example, that an expression in a PUT function must have the old, simple syntax. That is, we must have

```
LET PUT( LLL Basic Expression)
    = Expression
```

before the new parser will be called.

The patch itself is shown in Listing 2 (page 30). For more details of the operation of this patch refer to the pseudocode outline in Listing 1 (page 27). We hope this patch makes LLL Basic more useful for you.

■■J

(Listing begins on page 27)

LET Processor

(Listing 1 continued, text begins on page 26)

```
        IF The next character is a right parenthesis THEN
            PERFORM Process_Right_Parenthesis
        ELSE CALL SYMSRT to search the operator symbol table and re-
                  turn the operator index. The index is set to negative
                  one if the character is not found in the symbol table.
            IF The operator index returned does not indicate addition,
               subtraction, multiplication, or division THEN
                Syntax error is indicated
            ELSE PERFORM Process_Operator_Index
    END

    PROCEDURE Process_Left_Parenthesis
        The operator stack pointer is incremented.
        The left parenthesis identifier is stored on the operator stack
        The left parenthesis counter is incremented
    END

    PROCEDURE Process_Unary_Minus
        IF The most recent operator stored on the operator stack is the unary
           minus THEN
            Syntax error is indicated
        ELSE The operator stack is incremented
            The unary minus identifier is stored on the operator stack
    END

    PROCEDURE Process_Right_Parenthesis
        The left parenthesis counter is decremented.
        IF The left parenthesis counter is less than zero THEN
            Syntax error is indicated
        ELSE DO WHILE The operator identifier on top of the operator stack
                      is not the left parenthesis identifier
                CALL Unstack_Perform_Op
            The operator stack pointer is decremented
            The source line character pointer is incremented
            The source line character count is decremented
    END
```

(Continued on top of page 29)

```
PROCEDURE Process_Operator_Index
    Determine the new operator identifier from the operator index
    DO WHILE The precedence value of the operator identifier on top of
             the operator stack is greater than or equal to the precedence
             value of the new operator identifier
        CALL Unstack_Perform_Op
    The operator stack pointer is incremented
    The new operator identifier is stored on the operator stack
END

SUBROUTINE Unstack_Perform_Op
    Get the operator identifier on top of the operator stack
    The operator stack pointer is decremented
    IF The operator identifier is the unary minus identifier THEN
        The sign of the operand on top of the operand stack is inverted
    ELSE The second floating point register is set equal to the operand on
         top of the operand stack
        The operand stack pointer is decremented
        The first floating point register is set equal to the operand on top of
        the operand stack
        The required operation is determined from the operator identifier
        CALL BINOP to perform the indicated operation on the two
             floating point registers
        Store the result of the operation back on the operand stack
END

PROCEDURE Cleanup_Processing
    IF The left parenthesis counter is not equal to zero THEN
        Syntax error is indicated
    ELSE DO UNTIL The operator identifier on top of the operator stack
                  is the empty stack identifier
            CALL Unstack_Perform_Op
        The EXPRESSION_VALUE is set equal to the operand value on
        top of the operand stack
END
```

End Listing 1.

(Listing 2 begins on page 30)

LET Processor

(Listing continued, text begins on page 26)

Listing 2.

```
                        TABMSB  EQU     040Q            ; MSB of VALTAB start address
                        LPCNTR  EQU     20404Q          ; Left parenthesis counter register
                        OPRSP   EQU     20411Q          ; Operator stack pointer
                        OPRPTR  EQU     20405Q          ; Operator stack pointer register
                        OPDSP   EQU     20431Q          ; Operand stack pointer
                        OPDPTR  EQU     20407Q          ; Operand stack pointer register
                        LPREC   EQU     060Q            ; Left parenthesis precedence value

                        ; Operator precedence processor for LLL Basic LET statement right sides.
                                ORG     20000Q
20000 020               VALTAB: DB      020Q            ;
20001 021                       DB      021Q            ;
20002 042                       DB      042Q            ;
20003 043                       DB      043Q            ;
20004 257               EXEVAL: XRA     A               ; Initialize the
20005 062 004 041               STA     LPCNTR          ;       left parenthesis counter to zero.
20010 345                       PUSH    H               ; Save the source line character pointer
20011 075                       DCR     A               ; Let -1 be the operator stack bottom marker
20012 041 011 041               LXI     H,OPRSP         ; Get the operator stack pointer and place
20015 167                       MOV     M,A             ;       the bottom marker on the stack
20016 042 005 041               SHLD    OPRPTR          ; Store the pointer in the operator SP register
20021 056 031                   MVI     L,OPDSP AND 377Q ; Initialize the operand stack pointer
20023 042 007 041               SHLD    OPDPTR          ; Store the pointer in the operand SP register

                        ; Fetches next source line character. Tests to see if it is left parenthesis.
20026 341               L1:     POP     H               ; Restore source line character pointer
20027 315 373 025               CALL    ICP8            ; Bump pointer. If EOL, annunciate illegal expr.
20032 076 250                   MVI     A,250Q          ; Is new source line character
20034 276                       CMP     M               ;       a left parenthesis?
20035 302 055 040               JNZ     NOPAR           ; If not, go see what it is
20040 345                       PUSH    H               ; It is a left parenthesis. Save character pointer
20041 076 060                   MVI     A,LPREC         ; Assign it ID = 060, hence precedence value = 3
20043 041 004 041               LXI     H,LPCNTR        ; Bump the left
20046 064                       INR     M               ;       parenthesis counter.
20047 052 005 041               LHLD    OPRPTR          ; Fetch the operator stack pointer
20052 303 166 040               JMP     PUSHOP          ; Go push the left paren. to the OPR stack

                        ; Tests to see if character is a unary minus. If operand, it pushes value.
20055 076 255           NOPAR:  MVI     A,255Q          ; Is new source line character
20057 276                       CMP     M               ;       a unary minus?
20060 312 276 040               JZ      UMIN            ; If it is, go process it.
20063 315 324 027               CALL    VALUE           ; If operand, fetch value. If not, annunciate error.
20066 345                       PUSH    H               ; Save source line character pointer
20067 052 007 041               LHLD    OPDPTR          ; Fetch operand stack pointer and
20072 315 271 027               CALL    COPDH           ;       go push operand value to stack.
20075 076 004                   MVI     A,4             ; Bump the
20077 205                       ADD     L               ;       operand stack
20100 157                       MOV     L,A             ;       pointer and
20101 042 007 041               SHLD    OPDPTR          ;       store pointer in operand SP register.
20104 341                       POP     H               ; Restore source line character pointer
```

```
                        ; Tests to see if character is a right parenthesis or an operator.
20105 171               L2:     MOV     A,C     ; Get the source 'line remaining' counter and
20106 267                       ORA     A       ;       test for EOL. If at EOL,
20107 312 353 040               JZ      FINI    ;       go finish up the subroutine.
20112 076 251                   MVI     A,251Q  ; Is present source line character
20114 276                       CMP     M       ;       a right parenthesis?
20115 312 312 040               JZ      RPAR    ; If so, go compute value of parenthetical expr.
20120 076 002                   MVI     A,2     ; Not right paren. Set A = 2 and
20122 014                       INR     C       ;       bump C prior to calling SYMSRT so
20123 315 367 022               CALL    SYMSRT  ;       SYMSRT will search KDAT3 for operator.
20126 326 011                   SUI     9       ; Did SYMSRT find an operator?
20130 372 070 032               JM      ER8     ; If not, annunciate illegal expression

                        ; Compares operator precedence against top stack operator.
20133 345               OPR:    PUSH    H       ; Save the source line character pointer
20134 046 040                   MVI     H,TABMSB; The operator value from KDAT3 will index
20136 157                       MOV     L,A     ;       VALTAB to obtain operator precedence I.D.
20137 176                       MOV     A,M     ; Fetch operator precedence I.D.
20140 107                       MOV     B,A     ; Save it in B
20141 052 005 041               LHLD    OPRPTR  ; Fetch operator stack pointer
20144 176               OPR1:   MOV     A,M     ; Fetch precedence I.D. of top stack operator
20145 346 360                   ANI     360Q    ; Compute precedence value of top stack opr.
30147 137                       MOV     E,A     ; Save it in E
20150 170                       MOV     A,B     ; Restore new operators precedence I.D.
20151 346 360                   ANI     360Q    ;       and compute its precedence value
20153 273                       CMP     E       ; Compare its precedence to stack operators.
20154 170                       MOV     A,B     ; Restore precedence I.D. of new operator
20155 332 166 040               JC      PUSHOP  ; If new prec. > stack prec., go push it to stack
20160 315 176 040               CALL    USTAK   ; New prec.< stack prec. so go POP stack operator
20163 303 144 040               JMP     OPR1    ; Go back and test new opr. against next stack opr.

                        ; Pushes new operator to stack
20166 043               PUSHOP: INX     H       ; Bump operator stack pointer
20167 167                       MOV     M,A     ; Push new operator precedence I.D.
20170 042 005 041               SHLD    OPRPTR  ; Save operator stack pointer
20173 303 026 040               JMP     L1      ; Go deal with next source line character

                        ; Pops operator at top of operator stack
20176 107               USTAK:  MOV     B,A     ; Save accumulator
20177 257                       XRA     A       ; Is top stack operator
20200 276                       CMP     M       ;       a unary minus?
20201 302 220 040               JNZ     TUOPDS  ; If not, go do binary operation

                        ; Handles unary minus
20204 052 007 041       WUNOPD  LHLD    OPDPTR  ; Fetch the operand stack pointer
20207 053                       DCX     H       ; Point to characteristic of top stack operand
20210 176                       MOV     A,M     ; Fetch characteristic
20211 027                       RAL             ; Fetch its sign bit
20212 077                       CMC             ; Complement (negate) it
20213 037                       RAR             ; Restore sign bit (negate operand)
20214 167                       MOV     M,A     ; Restore characteristic
20215 303 265 040               JMP     OPOVER  ; Go wrap up the subroutine
```

(Continued on page 34)

LET Processor

(Listing 2 continued, text begins on page 26)

```
                        ; Perform binary operation on two operands at top of stack.
20220 353               TUOPDS: XCHG                    ; Save operator stack pointer in DE
20221 052 007 041               LHLD    OPDPTR          ; Fetch operand stack pointer
20224 175                       MOV     A,L             ; Two operands will be popped
20225 326 004                   SUI     4               ;       during the binary operation
20227 157                       MOV     L,A             ;       to follow so adjust the
20230 042 007 041               SHLD    OPDPTR          ;       operand SP accordingly.
20233 353                       XCHG                    ; Restore operator SP to HL
20234 176                       MOV     A,M             ; Top stack operator to A
20235 041 200 060               LXI     H,FREG2         ; Copy the top stack operand
20240 315 271 027               CALL    COPDH           ;       into floating point register 2.
20243 056 174                   MVI     L,FREG1 AND 377Q ; Copy the second
20245 033                       DCX     D               ;       stack
20246 033                       DCX     D               ;       operand
20247 033                       DCX     D               ;       into
20250 033                       DCX     D               ;       floating
20251 315 271 027               CALL    COPDH           ;       point register 1.
20254 353                       XCHG                    ; Save adjusted operand SP in HL
20255 346 017                   ANI     17Q             ; Compute operator value prior to BINOP
20257 315 121 032               CALL    BINOP           ; Go compute FREG1 operator FREG2
20262 315 271 027               CALL    COPDH           ; Push the result to the operand stack

                        ; Cleans up after unary or binary operation. Adjusts operator stack pointer.
20265 170               OPOVER: MOV     A,B             ; Operation is complete so restore A
20266 052 005 041               LHLD    OPRPTR          ; The operator stack was
20271 053                       DCX     H               ;       popped so adjust
20272 042 005 041               SHLD    OPRPTR          ;       the operator SP
20275 311                       RET                     ;       and return.

                        ; Makes sure that two consecutive unary minuses do not occur. Pushes minus to stack.
20276 345               UMIN:   PUSH    H               ; Save source line character pointer
20277 052 005 041               LHLD    OPRPTR          ; Fetch the operator stack pointer
20302 257                       XRA     A               ; Test the top stack operator
20303 276                       CMP     M               ; Is it also a unary minus?
20304 312 070 032               JZ      ER8             ; If it is, go annunciate illegal expression
20307 303 166 040               JMP     PUSHOP          ; Go push the unary minus to the operator stack

                        ; Computes value of parenthetical expression.
20312 345               RPAR:   PUSH    H               ; Save the source line character pointer
20313 041 004 041               LXI     H,LPCNTR        ; Fetch the left parenthesis counter
20316 065                       DCR     M               ; A left paren. is being popped
20317 372 070 032               JM      ER8             ; If no matching left paren., go annunciate error
20322 052 005 041               LHLD    OPRPTR          ; Fetch the operator SP
20325 076 060                   MVI     A,LPREC         ; Is the top stack operator

20327 276               RPAR1:  CMP     M               ;       a left parenthesis?
20330 312 341 040               JZ      LPAR            ; If so, go process it
20333 315 176 040               CALL    USTAK           ; Left paren. not yet exposed. Go perform operation
20336 303 327 040               JMP     RPAR1           ; Go back for next stack operator
```

```
                        ; Cleans up after evaluation of parenthetical expression.
20341 053               LPAR:   DCX     H               ; Left paren. popped so adjust operator SP
20342 042 005 041               SHLD    OPRPTR          ; Restore adjusted SP to operator SP register
20345 341                       POP     H               ; Restore source line character pointer to HL
20346 043                       INX     H               ; Bump it to point past the right paren.
20347 015                       DCR     C               ; Decrement the 'source line remaining' counter
20350 303 105 040               JMP     L2              ; Go deal with rest of source line

                        ; Completes processing of expression after EOL (source line) encountered.
20353 041 004 041       FINI:   LXI     H,LPCNTR        ; Fetch left parenthesis counter
20356 276                       CMP     M               ; Is it zero?
20357 302 070 032               JNZ     ER8             ; If not, too many left parentheses. Illegal
20362 075                       DCR     A               ; Routine will terminate when OPR stack empty
20363 052 005 041               LHLD    OPRPTR          ;       so fetch operator SP and
20366 276               FINI1:  CMP     M               ;       test for bottom of stack marker.
20367 312 000 041               JZ      FINIT           ; If found, go terminate routine
20372 315 176 040               CALL    USTAK           ; Not empty, so go perform next operation
20375 303 366 040               JMP     FINI1           ; Go back and look again for bottom marker

                        ; Finishes up routine.
20400 041 031 041       FINIT:  LXI     D,OPDSP         ; DE to point to expression value
20403 311                       RET                     ;       upon exiting this routine.
```

End Listing 2.

A Print Utility for CP/M

by Dan Daetwyler

Dan Daetwyler, Route 5, Box 518A, Springdale, AZ 72764.

I believe the only program that outproliferates YALE (Yet Another Line Editor) is a print utility. So why this one? Well, I like trees. In actual fact, I'm cheap. Paper, particularly printer paper, costs *money*. I find that I'm constantly listing and relisting the source of modules under development; having a printer that prints 132 columns, I got awfully annoyed at seeing half the page left blank. Hence a simple, double-column print utility. But like most "hackers" I couldn't leave well enough alone. I got tired of re-running the utility when I finally had a program cleaned up and wanted a final print. Setting up submit files was too much "busy work," etc., so I added a batching capability. Then, the CP/M ambiguous file name is useful, so why not include that too?

In short, a nice, simple little program turned into a 4K monster, but it has proven to be very handy for me. Hopefully, it will help you. Memory requirements are the 4K mentioned plus a page buffer for your size printer. This could be another whopping 7800 bytes, so if you're memory short, this isn't the program for you! If you're equipped with one of those nice printers (and exploiting software) that uses forms control, you need read further only from academic interest. This simple-minded beast is designed to work with dumb equipment. It ignores forms controls, handles its own pagination, and generally does things in a brute-force manner. My printer is a KSR-43 Teletype, which I might add is one fine machine. I've had it for several years and have never had a minute's problem with it. And that includes more cases of printer paper than I want my wife to find out about! This printer is a 132-character line, 51-line page machine, and that's what the utility is currently assembled to work with. If your page format is different, keep the faith. The program is "patchable," or may be re-assembled changing two constants (equates) and it will handle any reasonable format. I'll talk about installation a little later, so keep reading.

Listing Format

Hopefully, the listings accompanying this article are adequately commented. Rather than discuss the details of the code, this section will discuss some things that aren't included in the listings. I use one of the last versions of the defunct TDL (née ZITAN) assembler and linker, which I love. Fortunately, those tools are still available, since I believe they are one of the finest offerings available for the Z-80. Using this assembler/linker lets me break up my source into more manageable kernels and helps limit the re-assembly listing volume, so "DPRINT" is broken into 6 source files: DPRINT, DPR01, DPR02, DPR03, DPR04, DPRWS. I won't go into the functioning of .ENTRY and .EXTERN, which bring all of these pieces back together with addressability, but I will describe each of the macros and common I/O modules used by the print utility. Like many hackers I've quit reinventing the wheel every time I write a program. I have a set of common libraries that contain most of my primitives, and a set of macro libraries which simplify the calls to those modules. Since many of you will not have access to the TDL assembler/linker, and because of volume, I have chosen not to include the source listings of these materials. Rather, I will describe, in a very few words, each of the functions, and you may have the pleasure of mapping this thing into your I/O environment. First, I'll describe the limited set of macros used.

FCB Macro

This is a structure that defines a File Control Block (FCB) in CP/M's standard format, but with a four-byte header. The header contains a pointer to a file buffer for this file in the first two bytes, a buffer use count (current character offset) in the next byte, and the mode (input=1, output=2) in the fourth byte. Initially the use count is set to -1. All FCB's used by DPRINT are of this format.

SEARCH Macro

Sets up a call to the directory search module (common library), passing the type of search (first,next) and the address of the FCB. Returns with carry set if a "no find" condition occurs. Returns the standard CP/M offset (0-3) in the A register if a match is made.

OPEN Macro

Sets up a call to the common library open routine. Passes the FCB pointer and mode. An error exit is specified and taken if the file cannot be opened. The expansion used in this utility assumes the FCB is external to the macro and has been filled with name and type. Another expansion will generate an "in-line" FCB with name and type specified, but that is not used in this utility.

FGET Macro

Entry with a register pair containing the FCB address simply sets up the call to the module that will provide the next sequential character from an input file. Return is made with the character in the A register. An end-of-file exit is specified in the macro and taken if either a 1AH character is encountered or the CP/M read returns an end-of-file condition.

Common Library Modules Used

Although this list appears extensive, in actual practice the code for most of these modules is extremely primitive, simply providing the register save and restore necessary before invoking the usual CP/M function. The save and restore functions embedded in these calls restores all registers to their entry state except the A register.

$LO Module

Outputs the character contained in the A register to the system list device.

$CO Module

Same thing to the system console.

$LCRLF Module

Outputs a carriage return/line feed to the system list device.

$CRLF Module

Same thing to the system console.

$LBLK,$LBLKS Modules

$LBLK outputs a single blank to the system list device, while $LBLKS outputs as many blanks as the count in the A register specifies.

$BLK,$BLKS Modules

Same thing to the system console.

$MCI Module

Performs the CP/M function #10 (get input string). Expects the HL register pair to contain the address of a buffer, the A register to contain the maximum

length of permitted input plus two, and returns with the buffer filled and set up as described in your handy-dandy CP/M manual.

$MCPM Module

Performs the CP/M stringout function (function #9). Expects entry with the HL register pair pointing to a console message that is terminated with a $ and outputs this message to the system console.

$CIDR Module

Performs CP/M function #6 for input. Returns the result in the A register. This function reads the console input port without waiting and does not echo the key stroke.

$CI Module

Performs CP/M function #1, which is a simple read of the console input port. This function waits and echos. Return is made with the character in the A register.

$SERCH Module

Performs CP/M functions 17 and 18 (first,next) which search the file directory for the name specified in the FCB pointed to by the call. Returns with carry set if a "no find" condition occurs. Otherwise returns the A register content provided by CP/M.

$FGET Module

Simply provides the next sequential character from the file defined by the FCB on input. Return is made in the A register. Carry is set if either a 1AH character or a CP/M end-of-file condition is detected.

$OPEN Module

Open the file pointed to by the FCB pointer provided in the call. Sets the FCB prefix with the mode provided in the call, initializes the FCB (except for the drive, name and type fields, and sets the buffer-use byte to -1.

As you can see, these are all primitives that most of you who use CP/M have already developed to fit your own conventions. You can either substitute your macros and calls in this source material, or write simple primitives that are called directly by modifications you make in this code.

Installation

The assembled and linked program is a standard .COM file that may be easily patched for variations in printers. Locations 110H through 116H contain the following:

110H-111H	line length, least significant byte first
112H-113H	half of the line length, LSB
114H	size of paper (number of lines) *minus* 7
115H-116H	(line length) * (size of paper - 7), LSB

Location 117H, currently containing a RET (C9H), starts a 100H area that may be used for any special initialization code your configuration may require. This code will be entered once per invocation of DPRINT.

Operation

To run the utility you may initiate it by either of two forms of CP/M run commands. You may either enter:

```
DPRINT B:ANYFILE.TYP
```

(where B: is optional for the default disk)

or simply:

```
DPRINT
```

In either case, after load, the system will permit you to enter a constant to be printed on the heading line of each page of the listing produced. This can be anything you wish (not to exceed 40 characters), but was primarily intended for a date. If you do not wish a constant in the header, a null response is acceptable.

In case of the second form of the run command, the system will immediately start printing your file. If the file cannot be found, an open error message will be produced on the console. Paper positioning should be such that the first printable line on the page is positioned under the print head.

In the first form, the system will enter into interactive mode with you, prompting for the entry of the file names you wish to print. As you enter each name, the system will look it up in the file directory appropriate, display the result (either "can't find" or "print" followed by the file name), and wait for a key stroke. Any key other than N will cause the file to be stacked for printing.

Since ambiguous names are permitted, entries such as *.ASM or DPR???.-ASM may produce several possible file names for stacking, and each is presented to you for inclusion or discard. When all matching names have been found, the prompt for another file name is issued. To terminate this process, make a null entry. The system will stack up to 64 files for print, and then will automatically exit the interactive dialog.

Upon exit from the interactive dialog, the system will immediately begin to print the first file in the stack. During the printing process, you may interrupt the print by any of the following keys:

Control-S	Pauses print – any key will resume printing.
Control-C	Cancels total job and returns to CP/M.
Escape	Cancels print of the current file and proceeds to the next file (assuming stack is still occupied).

This interruption will occur at the current line. When a file or job cancellation occurs, the form is advanced to the top of a new page.

When the stack is exhausted, the program will prompt to see if you have further printing or not. If you respond Y to the prompt, the system will re-enter the interactive dialog phase, and the process iterates.

For those of you who do not want to poke all of this into a terminal, I'll make the standard offer. . . . If you send me a disk and $20, or $30 without a disk, I'll format it to single-density single-surface, and load it with the source, relocatable, and executable materials for DPRINT and my common libraries and macros. The latter probably won't do you much good unless you have the TDL (or equivalent) assembler and linker, but for those who do, it will save some code. Incidentally, the fee is to cover copying, shipping, and handling. I'm like Dr. Dobbs . . . I'm non-profit, but in my case it probably wasn't intentional! From previous experience, I'm a bit "gun shy." I use a Morrow Engineering 2D disk interface with Shugart 800 series disks. By going to single-density, single-surface, I've always been able to migrate materials, but because of those of you with "odd" interfaces, etc., this offer is without warranty. Incidentally, I'm on CP/M Version 2.x, but that should not cause problems.

Good luck, and happy printing!!

(Listing begins on page 39)

CP/M Print Utility (Text begins on page 36)

Listing 1

```
Listing of DPRINT.ASM      March 18, 1982

        .TITLE  "DOUBLE COLUMN PRINT OF DISK ASCII FILE - MAIN
                 MODULE"
        .SBTTL  "VERSION 2.0 - MARCH 1982 - DD SYSTEMS"
        .IDENT  DPRINT
;
        .EXTERN SETSTK                          ;DPR01
        .EXTERN PRELD,LDFCB                     ;DPR02
        .EXTERN CLEAR,PRINT                     ;DPR03
        .EXTERN FILL                            ;DPR04
        .EXTERN .FREE.                          ;SYSTEM
;
;                       *********************************
;                       *                               *
;                       * COPYRIGHT 1982  -  DD SYSTEMS *
;                       *      ALL RIGHTS RESERVED      *
;                       *                               *
;                       *********************************
;                       *                               *
;                       * D. W. DAETWYLER               *
;                       * ROUTE 5, BOX 518A             *
;                       * SPRINGDALE, ARK. 72764        *
;                       * 501-756-0212                  *
;                       *                               *
;                       *********************************
;
;       NOTE:  THIS COLLECTION OF MODULES IS RELEASED FOR
;              INDIVIDUAL, PRIVATE, USE.  ANY COMMERCIAL
;              USE OR EXPLOITATION IS EXPRESSLY FORBIDDEN.
;              "CP/M" IS A COPYRIGHTED DEVELOPMENT OF
;              DIGITAL RESEARCH.  "KSR-43" IS COPYRIGHTED
;              BY THE TELETYPE CORPORATION.
;
;       THIS PROGRAM PROVIDES A DOUBLE COLUMN PRINT OF TYPICAL
;       SOURCE (ASCII) FILES FOR ANY LANGUAGE.  IT IS DESIGNED
;       TO USE ANY REASONABLE PAGE SIZE (MAX 132 X 66) AND IS
;       CURRENTLY ASSEMBLED TO RUN WITH A KSR-43 TELETYPE. IT
;       HANDLES ITS OWN PAGINATION AND IGNORES FORMS CONTROLS.
;       PAGES ARE TITLED AND NUMBERED.  LINES TOO LONG (OVER
;       HALF PAGE - 2) TO FIT ARE BROKEN ON "TEXT WORD"
;       BOUNDARIES, AND THE CONTINUATION IS INDENTED 20
;       POSITIONS ON THE FOLLOWING LINE.
;
;       INPUT TO THE PROGRAM MAY BE IN THE CP/M COMMAND LINE
;       AS IS USUAL FOR UTILITIES (EG: DPRINT B:DUMMY.ASM).



;       ADDITIONAL OPTIONS ARE OFFERED, HOWEVER.  SIMPLY
;       ENTERING THE RUN COMMAND (EG: DPRINT) ALONE, WILL
;       CAUSE THE PROGRAM TO INTERROGATE THE CONSOLE FOR ITS
;       INPUT FILE OR FILES.  UP TO 64 FILES MAY BE SPECIFIED
;       DURING THIS INPUT SESSION.  END OF SESSION IS
;       SIGNIFIED BY A NULL RESPONSE (IMMEDIATE RETURN).
;
;       WHEN ENTERING FILES IN THIS MODE, THE CP/M CONVENTIONS
;       PERMITTING THE USE OF ???? OR * AS UNIVERSAL MATCH
;       CHARACTERS IS ALLOWED.
;
;       AS EACH FILE IS REQUESTED (MULTIPLE RESPONSE WITH
;       (???? OR *), IT IS LOOKED UP IN THE DISK DIRECTORY,
;       AND THE USER IS ASKED IF THIS FILE IS TO BE PRINTED.
;       ANY RESPONSE OTHER THAN "N" WILL PERMIT THE FILE TO
;       BE STACKED FOR PRINT.  "N" WILL BYPASS THAT FILE
;       AND CONTINUE THE PROCESS.  BY ENTERING *.ASM, THE
;       PROGRAM WILL PERMIT YOU TO SELECT FROM ALL OF THE
;       .ASM FILES ON THE DISK...SAVES A FEW KEY STROKES.
;
;       WHEN MULTIPLE FILES ARE PRINTED, THE HEADINGS AND PAGE
;       NUMBERING IS RE-INITIALIED FOR EACH FILE.  DURING
;       PRINTING, THE PROCESS MAY BE INTERUPTED AT THE END OF
;       ANY LINE. A "CONTROL-S" WILL PAUSE PRINTING UNTIL
;       ANY KEY IS DEPRESSED.  AN "ESCAPE" WILL CAUSE THE FILE
;       CURRENTLY BEING PRINTED TO BE TERMINATED, THE PAGE
;       EJECTED, AND THE NEXT FILE (ASSUMING ONE AWAITS) TO BE
;       STARTED.  A "CONTROL-C" ABORTS THE JOB AND RETURNS TO
;       THE SUPPORTING SYSTEM.
;
;       INITIAL ENTRY WILL REQUEST A "TITLE" THAT WILL APPEAR
;       ON THE HEADING LINE OF EACH PRINT OUT (ALONG WITH THE
;       FILE NAME AND PAGE NUMBER).  THIS "TITLE" DOES NOT
;       CHANGE DURING THE ENTIRE RUN, AND IS REALLY INTENDED
;       FOR INCLUSION OF A DATE.  A NULL ENTRY IS ACCEPTED.
;
;       FINALLY, THE PROGRAM, BEFORE TERMINATION, WILL ALLOW YOU
;       TO RE-ENTER THE FILE SELECTION DIALOG IF YOU DESIRE.
;
;       CHARACTERS LESS THAN BLANK, AND OTHER THAN CARRIAGE
;       RETURN, LINE FEED, BACK SPACE, AND TAB WILL PRINT AS THE
;       "UNPRINTABLE" CHARACTER SYMBOL ON THE PRINTER.
;
;


;               SYSTEM EQUATES
;
FCB     ==      5CH     ;DEFAULT SYSTEM FCB
BUF1    ==      80H     ;CP/M DEFAULT BUFFER
;
;       AREA FOR PAGE CONSTANTS
;
L.L     ==      132
P.L     ==      51
LLEN::  .WORD   L.L     ;LINE LENGTH
HLEN::  .WORD   L.L/2   ;HALF LINE LENGTH
LINES:: .BYTE   P.L-7   ;PAGE LENGTH - HEADER AND TRAILER
PROD::  .WORD   (P.L-7)*L.L-1 ;PAGE MATRIX SIZE
;
;       USER "PATCH" FOR INITIALIZATION
;
DUMMY:  RET
        .BLKB   127
;
;       ACTUAL START OF PROGRAM
;
START:  XRA     A
        STA     STKCNT# ;INITIALIZE WORKING STORAGE



        STA     PGCNT#
        LHLD    .FREE.  ;LOAD TOP OF PROGRAM
        LDED    PROD    ;LOAD PAGE SIZE-1
        DAD     D
        INX     H       ;HL POINTS TO LAST CELL USED
        LDED    6       ;LOAD BOTTOM OF OPERATING SYSTEM
        DCR     D       ;REDUCE FOR STACK
        XCHG
        ORA     A
        DSBC    D       ;SEE IF ENOUGH SPACE
        JRNC    ..XX
        LXI     H,PRM6
        CALL    $MCPM   ;ANNOUNCE MEMORY SHORTAGE
        JMP     0       ;AND SCRUB
..XX:   CALL    TITLE   ;GET HEADING FIELD
        LXI     H,STACK#
        SHLD    STKPTR#
        LDA     FCB+1   ;SEE IF FILE NAME WAS INPUT WITH COMMAND
        CPI     ' '
        JRZ     ..RPT   ;NO, SO ENTER CONSOLE DIAGLOG
;
;       FILE NAME ON COMMAND LINE
;
        LXI     H,FCB
        LXI     D,F$INP#+4      ;INPUT FCB
        LXI     B,32
```

(Continued on next page)

CP/M Print Utility (Listing continued, text begins on page 36)

```
        LDIR            ;SINCE CP/M SET IT UP, USE IT!
        CALL    PRELD   ;GO TO PRELOAD ENTRY POINT
        JRC     ..DNE   ;OPEN FAILURE
        JMPR    ..GO    ;AND REENTER PROCESS
;
;       CONSOLE DIALOG FOR FILE ENTRY
;
..RPT:  CALL    $CRLF#
        LXI     H,PRM1  ;PROMPT FOR DATA
        CALL    $MCPM#
        JMPR    ..B
..A:    LXI     H,BUF1  ;USE DEFAULT BUFFER FOR INPUT
        MVI     A,20    ;MAX INPUT=20 CHARACTERS
        CALL    $MCI#   ;GET INPUT STRING
        INX     H
        XRA     A
        CMP     M       ;CHECK FOR NULL INPUT
        JRZ     ..SDN   ;YEP, SO ALL DONE BUT THE PROCESSING
        CALL    SETSTK  ;ADD ENTRY TO STACK
        JRC     ..SDN   ;20 ENTRIES, SO PUNT
..B:    CALL    $CRLF#
        LXI     H,PRM2  ;CONTINUE PROMPTING
        CALL    $MCPM#
        JMPR    ..A
;
;       HAVING THE INPUT LIST, NOW PROCESS
;
..SDN:  CALL    $CRLF#
        LXI     H,PRM4
        CALL    $MCPM#  ;PROMPT FOR PRINT PHASE
        LXI     H,STACK#
        SHLD    STKPTR# ;RESET THE STACK POINTER TO START
..CHK:  CALL    LDFCB   ;SET UP THE FCB AND OPEN
        JRC     ..DNE   ;STACK IS EXHAUSTED
..GO:   XRA     A
        STA     PGCNT#  ;RESET PAGE NUMBER
        LXI     Y,LINSX#
        MVI     0(Y),0  ;INITIALIZE LINE CONTROL SWITCH
..G1:   CALL    CLEAR   ;CLEAR THE PAGE PRINT AREA
        CALL    FILL    ;FILL IT WITH DATA
        JRC     ..EOF   ;END OF FILE
        CALL    PRINT   ;NOW LIST IT
        JRC     ..TST   ;SOMEONE IS TALKING TO US
        JMPR    ..G1    ;LOOP THROUGH FILE
..EOF:  CALL    PRINT   ;STILL HAVE TO PRINT THAT LAST PAGE
        JRC     ..TST
        JMPR    ..CHK   ;NO, SO GO FOR NEXT STACK ENTRY
;
;       THE STACK IS EXHAUSTED (AND SO AM I)
;
..DNE:  LXI     H,PRM3  ;SEE IF HE WANTS TO PRINT THE PHONE BOOK
        CALL    $CRLF#
        CALL    $MCPM#
        CALL    $CI#    ;JUST ONE CHARACTER IS ENOUGH
        CPI     'Y'     ;IF HE FUMBLES THIS, HE'S OFF
        JNZ     0
        XRA     A
        STA     STKCNT#
        LXI     H,STACK#
        SHLD    STKPTR#
        JMPR    ..RPT   ;HERE WE GO AGAIN!!
;
;       USER ENTERED A CANCEL BATCH OR CANCEL FILE COMMAND
;
..TST:  DCR     A
        JRZ     ..CHK   ;ABORTED THIS FILE
        JMP     0       ;ABORTED THE BATCH
;
;       HEADER LINE CONSTANT ENTRY
;
TITLE:  CALL    DUMMY   ;DUMMY (RET) PATCH FOR SPECIAL
                            INITIALIZATION
        LXI     H,PRM5
        CALL    $MCPM#  ;PROMPT FOR HEADING
        LXI     H,BUF2#
        MVI     A,42
        CALL    $MCI#   ;GET USER INPUT
        INX     H
        LXI     B,0
        MVI     C,40    ;MAX ACCEPTED
        MOV     A,M
        CPI     41
        JRNC    ..A
        MOV     C,A     ;LESS, SO USE HIS LENGTH
..A:    INR     C       ;BUMP EITHER BY ONE
        LXI     D,TILSAV#
        LDIR            ;SAVE TITLE
        RET
;
;       LOCAL MESSAGES
;
PRM1:   .BYTE   1BH,'E' ;CLEARS SCREEN ON SOME TERMINALS
        .ASCII  "                                   FILE ENTRY"
        .BYTE   13,10,13,10
        .ASCII  "ENTER FILE NAMES IN CONVENTIONAL FORM:"
        .BYTE   13,10
        .ASCII  "       [DRIVE:]NAME.TYPE"
        .BYTE   13,10
        .ASCII  "AMBIGUOUS REFERENCES (????? OR *) ARE
                    SUPPORTED.  LIMIT IS"
        .BYTE   13,10
        .ASCII  "64 FILES FOR PRINT.  ENTER A NULL (ZERO LENGTH)
                    ENTRY TO"
        .BYTE   13,10
        .ASCII  "TERMINATE THE SELECTION PROCESS."
        .BYTE   13,10,'$'
PRM2:   .ASCII  "FILE? (NULL TO QUIT): $"
PRM3:   .ASCII  "MORE (Y OR N)? $"
PRM4:   .ASCII  "'ESCAPE' TERMINATES PRINT OF FILE CURRENTLY
                    PRINTING"
        .BYTE   13,10
        .ASCII  "'CONTROL-S' PAUSES PRINTING - ANY KEY RESUMES"
        .BYTE   13,10
        .ASCII  "'CONTROL-C' TERMINATES AND RETURNS CONTROL TO
                    THE SYSTEM"
        .BYTE   13,10,'$'
PRM5:   .ASCII  "ENTER HEADING: $"
PRM6:   .ASCII  "SORRY - NOT ENOUGH MEMORY FOR THIS PAGE
                    SIZE!!$"
;
        .END    START
```

Listing 2

```
Listing of DPR01.ASM      March 18, 1982

        .TITLE  "DOUBLE COLUMN PRINT - SEARCH STACK MODULES"
        .SBTTL  "VERSION 2.0 - MARCH 1982 - DD SYSTEMS"
        .IDENT  DPR01
;
        .ENTRY  SETSTK
;
        .EXTERN LOGFIL
;
        .INSERT MACF1
;
;       THIS SET OF MODULES ACCEPTS THE INPUT STRING FROM THE
;       USER AND FORMS IT INTO "FCB" FORM.  '*' AND '????' ARE
;       HANDLED AS ARE "DRIVE" SPECIFICATIONS.
```

```
;
;       AS EACH NAME IS ACCEPTED, IT IS USED FOR A DIRECTORY
;       SEARCH.  EACH SATISFIED SEARCH IS DISPLAYED, AND THE
;       USER IS PERMITTED TO ACCEPT/DISCARD THE ENTRY.  IF
;       THE ???? OR * MATCH CHARACTERS ARE USED, THEN ALL OF
;       THE FILES MEETING THESE CRITERIA ARE DISPLAYED FOR
;       USER SELECTION.  ACCEPTED FILES ARE STACKED FOR LATER
;       PROCESSING.  IF A NAME CANNOT BE FOUND, THEN THAT
;       INCIDENT IS LOGGED, AND THE PROCESS CONTINUES.
;
SETSTK: LXI     D,F$SRCH#+4 ;POINT TO SEARCH FCB
        MOV     B,M         ;GET STRING LENGTH
        INX     H           ;STEP TO DATA
        CALL    SETDRV      ;TRY TO FILL IN THE DRIVE SPEC
        CALL    SETNAM      ;MOVE IN THE FILE NAME
        CALL    SETTYP      ;MOVE IN THE FILE TYPE
        SEARCH  SRCH,FIRST
        JRC     ..B         ;NOT FOUND
..A:    CALL    DSPFND      ;DISPLAY FILE FOUND
        JRC     ..C         ;USER BYPASSED
        CALL    ADSTAK      ;ADD ENTRY TO STACK
        RC
..C:    SEARCH  SRCH,NEXT
        JRNC    ..A         ;GOOD HIT
        XRA     A           ;NO FIND, BUT SECONDARY
        RET
..B:    CALL    NOFND       ;PRIMARY SEARCH NOT FOUND
        XRA     A
        RET
;
;       SEE IF A DRIVE WAS SPECIFIED, OR IF IT IS A "DEFAULTED"
                            FILE
;
SETDRV: PUSH    H
        PUSH    D
        PUSH    B
        PUSH    D
        POP     H
        INX     D
        MVI     M,' '
        LXI     B,11
        LDIR                ;BLANK THE STACK ENTRY AREA
        LXI     B,24
        MVI     M,0         ;CLEAR THE REMAINDER OF THE FCB
        LDIR
        POP     B
        POP     D
        POP     H
        XRA     A
        STAX    D           ;SET AS "DEFAULT"
        INX     D
..A:    MOV     A,M
        INX     H
        CPI     ' '
        JRNZ    ..X         ;STRIP LEADING BLANKS IF ANY
        DJNZ    ..A
..X:    MOV     A,M
        CPI     ':'         ;CHECK FOR DRIVE FLAG
        DCX     H
        RNZ                 ;DEFAULT DRIVE
        MOV     A,M
        SUI     'A'
        RC                  ;BAD DRIVE, SO DEFAULT
        INR     A
        DCX     D
        STAX    D           ;SET IN DRIVE CODE
        INX     D
        INX     H
        INX     H
        DCR     B

        DCR     B
        RET
;
;       NOW FIT THE NAME INTO THE FCB
;
SETNAM: PUSH    D
        XRA     A
        CMP     B
        JRZ     ..RET       ;NO NAME - WELL OPEN WILL GET HIM
..A:    MOV     A,M
        INX     H
        CPI     '*'
        JRZ     ..UN        ;UNIVERSAL MATCH - WOW!
        CPI     '.'
        JRZ     ..END       ;FOUND END OF NAME
        STAX    D
        INX     D
        DJNZ    ..A
..RET:  POP     D
        XCHG
        PUSH    B
        LXI     B,8
        DAD     B
        POP     B
        XCHG                ;DE STEPPED OVER NAME FIELD
        RET

;
;       '*' MUST BE CONVERTED TO ????????
;
..UN:   POP     D           ;RESTORE PTR TO START OF FIELD
        PUSH    D
        XCHG
        PUSH    B
        MVI     B,8
..B:    MVI     M,'?'
        INX     H
        DJNZ    ..B

        POP     B
        XCHG
        INX     H
        DCR     B
..END:  DCR     B
        JMPR    ..RET
;
;       NOW SET THE FILE TYPE
;
SETTYP: PUSH    D
        XRA     A
        CMP     B



        JRZ     ..RET
..A:    MOV     A,M
        INX     H
        CPI     '*'
        JRZ     ..UN        ;UNIVERSAL MATCH AGAIN
        STAX    D
        INX     D
        DJNZ    ..A
..RET:  POP     D
        RET
..UN:   POP     H
        MVI     B,3
..B:    MVI     M,'?'
        INX     H
        DJNZ    ..B
        RET
;
;       DISPLAY THE FILE NAME FOUND AND GET ACCEPTANCE
;
DSPFND: CALL    SETRD       ;SET UP NAME FROM DIRECTORY
        CALL    $CRLF#
        LXI     H,FMSG      ;PROMPT FOR ACCEPTANCE
        CALL    $MCPM#
        LXI     H,SAVFIL
```

(Continued on next page)

CP/M Print Utility (Listing continued, text begins on page 36)

```
        CALL    LOGFIL
        MVI     A,'?'
        CALL    $CO#
        CALL    $BLK#
        CALL    $CI#    ;GET INPUT RESPONSE
        CPI     'N'     ;ONLY 'N' REJECTS
        STC
        RZ

        XRA     A
        RET
;
;       GET DIRECTORY ENTRY AND MOVE TO SAVE AREA
;
SETRD:  ADD     A
        ADD     A
        ADD     A
        ADD     A
        ADD     A
        LXI     B,0
        MOV     C,A




        LXI     H,BUF2#
        DAD     B       ;HL POINTS TO DIRECTORY ENTRY
        LXI     D,SAVFIL
        LXI     B,12
        LDIR            ;SAVE NAME
        LDA     F$SRCH#+4 ;GET DRIVE
        STA     SAVFIL
        RET
;
;       ADD SAVE ENTRY TO STACK
;
ADSTAK: LDED    STKPTR#
        LXI     H,SAVFIL
        LXI     B,12
        LDIR
        SDED    STKPTR# ;SAVE UPDATED POINTER
        LDA     STKCNT#
        INR     A       ;INCREMENT STACK COUNT
        STA     STKCNT#
        CPI     64
        CMC
        RET
;
;       PRIMARY ENTRY NOT FOUND, SO LOG IT
;
NOFND:  CALL    $CRLF#
        LXI     H,NFMSG
        CALL    $MCPM#
        LXI     H,F$SRCH#+4
        CALL    LOGFIL
        RET
;
;       LOCAL MESSAGES
;
FMSG:   .ASCII  "Print $"
NFMSG:  .ASCII  "Can't find $"
;
;       LOCAL WORK AREA
;
SAVFIL: .BLKB   12
;
        .END
```

Listing 3

```
Listing of DPR02.ASM      March 18, 1982

        .TITLE  "DOUBLE COLUMN PRINT - SET UP FCB'S AND OPEN"
        .SBTTL  "VERSION 2.0 - MARCH 1982 - DD SYSTEMS"
        .IDENT  DPR02
;
        .ENTRY  LDFCB,PRELD,LOGFIL
;
        .INSERT MACF1
;
;       THIS MODULE PULLS FILE NAMES FROM THE STACK AND OPENS
;       THEM.  "PRELD" IS AN ENTRY USED WHEN THE FILE IS
;       SPECIFIED ON THE CP/M COMMAND LINE.  FILE NAMES
;       ARE PLACED IN THE INPUT FCB FOR OPEN.  IN THE RARE
;       CASE OF AN OPEN FAILURE, THE INCIDENT IS LOGGED, AND
;       THE RUN CONTINUES.
;
;       SYSTEM EQUATES
;
BUF1    ==      80H     ;CP/M DEFAULT BUFFER
;
;       PULL NAME FROM STACK, DO INITIAL DIRECTORY SEARCH, AND
                OPEN
;
LDFCB:  LDA     STKCNT#
        ORA     A       ;CHECK THE STACK COUNT
        STC
        RZ              ;STACK EXHAUSTED
        DCR     A
        STA     STKCNT#
        LHLD    STKPTR# ;GET NAME ADDRESS
        LXI     D,F$INP#+4 ;POINT TO SEARCH FCB
        LXI     B,12
        LDIR            ;MOVE IN NAME (AND DRIVE)
        SHLD    STKPTR# ;SAVE THE UPDATED STACK PTR
PRELD:  OPEN    INP,INPUT,,,,BUF1,OPERR
        XRA     A
        RET
;
;       OPEN ERROR LOGGING
;
OPERR:  CALL    $CRLF#
        LXI     H,OMSG
        CALL    $MCPM#  ;START MESSAGE
        LXI     H,F$INP#+4
        CALL    LOGFIL  ;OUTPUT FILE NAME




        JMPR    LDFCB
;
;       OUTPUT FILE NAME TO CONSOLE
;
LOGFIL: MOV     A,M
        ORA     A       ;CHECK DRIVE
        JRZ     ..A     ;DEFAULTED
        ADI     'A'-1
        CALL    $CO#    ;OUTPUT DRIVE
        MVI     A,':'
        CALL    $CO#    ;AND FLAG



..A:    INX     H
        MVI     B,8
..C:    MOV     A,M
        INX     H
        CPI     ' '
        JRZ     ..B     ;SKIP BLANKS
        CALL    $CO#
..B:    DJNZ    ..C     ;LOOP THROUGH NAME
        MOV     A,M
        CPI     ' '     ;SEE IF THERE'S A TYPE
        RZ              ;NO, SO GET OUT
        MVI     A,'.'
        CALL    $CO#
        MVI     B,3
..D:    MOV     A,M
```

```
        INX     H
        CALL    $CO#
        DJNZ    ..D
        RET
;
;       LOCAL MESSAGE
;
OMSG:   .ASCII  "CAN'T OPEN $"
;
        .END
```

Listing 4

```
Listing of DPR03.ASM      March 18, 1982

        .TITLE  "DOUBLE COLUMN PRINT - CLEAR AND PRINT MODULES"
        .SBTTL  "VERSION 2.0 - MARCH 1982 - DD SYSTEMS"
        .IDENT  DPR03
;
        .ENTRY  CLEAR,PRINT
;
        .EXTERN .FREE.



;
;       THIS MODULE CLEARS THE PAGE BUFFER AREA AND THEN PRINTS
;       IT AFTER IT HAS BEEN SET UP BY "FILL".  IT SAMPLES THE
;       KEY BOARD DURING PRINTING FOR POSSIBLE INTERRUPTIONS,
;       AND TAKES APPROPRIATE ACTION ON VALID INPUT.
;



CLEAR:  LHLD    .FREE.  ;USES MEMORY ABOVE PROGRAM
        PUSH    H
        POP     D
        INX     D
        LBCD    PROD#   ;PAGE SIZE
        MVI     M,' '
        LDIR            ;CLEARED TO BLANKS
        RET



;
;       PRINTS THE PAGE BUFFER
;


PRINT:  CALL    $LCRLF#
        CALL    $LCRLF# ;SPACE DOWN FROM TOP
        CALL    $LCRLF#
        CALL    HEAD    ;PRINT THE HEADING

        LHLD    .FREE.
        LDED    LLEN#   ;LOAD LINE LENGTH
        LDA     LINES#  ;LOAD LINE COUNT
        MOV     B,A

..LP:   CALL    PRTLNE  ;PRINT A LINE
        DAD     D
        CALL    $CIDR#  ;CHECK FOR KEY PRESS
..XX:   ORA     A
        JRZ     ..A
        CPI     1BH
        JRZ     ..PNT1  ;PUNT THIS FILE
        CPI     3
        JRZ     ..PNT2  ;PUNT THE JOB
        CPI     13H     ;CONTROL-S
        JRNZ    ..A
        CALL    $CI#    ;WAIT FOR KEY PRESS



        JMPR    ..XX    ;NOW SEE IF IT WAS A CONTROL KEY
..A:    DJNZ    ..LP
        CALL    $LCRLF# ;FINISH THE PAGE
        CALL    $LCRLF#
        XRA     A
        RET


..PNT1: MVI     A,1
..PNT2: PUSH    PSW     ;SAVE RETURN CODE
        INR     B
..PNT3: CALL    $LCRLF# ;SPACE OUT PAGE
        DJNZ    ..PNT3
        POP     PSW
        STC
        RET



;
;       PRINT A PAGE LINE
;
PRTLNE: PUSH    H
..LP:   MOV     A,M
        CPI     13
        JRZ     ..CR    ;QUIT ON CR
        CALL    $LO#
        INX     H
        JMPR    ..LP



..CR:   CALL    $LCRLF# ;END LINE
        POP     H
        RET


;
;       PRINT THE PAGE HEADING
;
HEAD:   LXI     H,HMS1
        CALL    LMSG
        LXI     H,F$INP#+4
        MVI     B,8
        MOV     A,M
        INX     H
        ORA     A
        JRZ     ..A     ;CHECK DRIVE
        ADI     'A'-1
        CALL    $LO#
        MVI     A,':'



        CALL    $LO#    ;DRIVE FLAG
..A:    MOV     A,M
        INX     H



        CPI     ' '
        JRZ     ..B     ;SKIP BLANKS
        CALL    $LO#
```

(Continued on next page)

```
..B:    DJNZ    ..A
        MOV     A,M
        CPI     ' '     ;CHECK FOR TYPE
        JRZ     ..C
        MVI     A,'.'



        CALL    $LO#    ;PUNCTUATE
..C:    MVI     B,3
..D:    MOV     A,M
        INX     H
        CALL    $LO#    ;OUTPUT TYPE
        DJNZ    ..D
        MVI     A,6
        CALL    $LBLKS# ;OUTPUT 93 SPACES



        LXI     H,TILSAV+1
        CALL    LMSG    ;OUTPUT USER HEADER
        LDA     LLEN#   ;GET LINE LENGTH
        SUI     78



        CALL    $LBLKS# ;SPACE OVER TO PAGE NUMBER
        LXI     H,HMS2
        CALL    LMSG
        LDA     PGCNT#  ;GET PAGE COUNT
        INR     A
        STA     PGCNT#
        MVI     B,0
..F:    SUI     10
        JRC     ..E
        INR     B
        JMPR    ..F
..E:    ADI     10
        MOV     C,A
        XRA     A
        CMP     B
        MVI     A,' '
        JRZ     ..G
        MVI     A,'0'


..G:    ORA     B
        CALL    $LO#    ;OUTPUT PAGE NUMBER
        MVI     A,'0'
        ORA     C
        CALL    $LO#
        CALL    $LCRLF#



        CALL    $LCRLF#
        RET
;
;       DUMB LIST MESSAGE ROUTINE
;
LMSG:   MOV     A,M
        ORA     A
        RZ
        INX     H
        CALL    $LO#
        JMPR    LMSG



;
;       LOCAL MESSAGES
;
HMS1:   .ASCIZ  "Listing of "
HMS2:   .ASCIZ  "Page "
;
TILSAV::.ASCIZ  "
;
        .END
```

Listing 5

```
        .TITLE  "DOUBLE COLUMN PRINT - PAGE FILLER MODULES"
        .SBTTL  "VERSION 2.0 - MARCH 1982 - DD SYSTEMS"
        .IDENT  DPR04
;
        .ENTRY  FILL
;
        .EXTERN .FREE.
;
        .INSERT MACF1


;
;       THESE MODULES PROVIDE THE FUNCTION OF BUILDING THE TWO
;       COLUMN SPREAD IN A PAGE SIZE BUFFER.  THE "FILL"
;       MODULE IS SIMPLE OVERHEAD THAT ITERATES THROUGH THE
;       "COL" ROUTINE TWICE.  THE "COL" ROUTINE, AGAIN, IS
;       OVERHEAD THAT CYCLES THROUGH THE "LINE" ROUTINE "LINES"
;       TIMES.  "LINE" IS THE REAL PROCESSOR, WHICH FORMS THE
;       LINE IN A BUFFER BY USE OF "GETLIN" AND THEN MOVES
;       APPROPRIATE PARTS (LONG LINES) OF THE LINE TO THE PAGE
;       AREA.  "GETLIN" READS THE DISK AND BUILDS A FULL LINE
;       (UP TO 255 BYTES) IN THE BUFFER AREA.  IT READS UNTIL
;       A CR IS SENSED.  FINALLY, "SCAN" (CALLED BY "FILL")
;       SCANS THE PAGE BUFFER AND ESTABLISHES CARRIAGE RETURN/
;       LINE FEEDS AT THE APPROPRIATE POINTS IN THE LINE.
;       NOTE:  THIS ROUTINE WILL NOT HANDLE PAGES OF WIDTHS
;       OF 510 COLUMNS OR GREATER (NOT THAT ANY ONE USES SUCH
;       A PRINTER).
;
;
;       PAGE FILL
;



FILL:   LIXD    .FREE.
        CALL    COL     ;FORM FIRST COLUMN
        JRC     ..A     ;EOF
        LIXD    .FREE.
        LDED    HLEN#
        DADX    D       ;OFFSET HALF PAGE
        CALL    COL     ;FORM SECOND COLUMN
..A:    PUSH    PSW     ;SAVE FLAGS
        CALL    SCAN    ;SET CR/LF
        POP     PSW     ;RESTORE FLAGS
        RET
;
;       BUILD ONE COLUMN
;



COL:    LDA     LINES#
        MOV     B,A     ;LINE COUNT
        LDED    LLEN#   ;LINE LENGTH
..A:    CALL    LINE    ;BUILD A LINE
        RC              ;EOF RETURN
        DADX    D
        DJNZ    ..A
        XRA     A
        RET
;
;       BUILD A LINE
;
LINE:   PUSH    D
        PUSH    B
        BIT     0,0(Y)  ;CHECK OVERFLOW FLAG
        JNZ     ..UGH   ;OVER FLOW WAITING
        CALL    GETLIN  ;GET THE NEXT LINE
        ORA     A
        JRZ     ..XX    ;ZERO LENGTH LINE (USUALLY EOF)
..GO:   MOV     C,A
```

```
        LDA     HLEN#   ;LOAD HALF LENGTH
        DCR     A
        DCR     A

        CMP     C
        JRC     ..LNG   ;LINE TOO LONG
        RES     0,0(Y)  ;CLEAR OVERFLOW FLAG
..MVE:  PUSH    X
        POP     D       ;DE POINTS TO LINE/COL IN PAGE
        LXI     H,LINBUF#
        MVI     B,0
        LDIR            ;MOVE IN LINE
..XX:   XRA     A
        BIT     1,0(Y)  ;CHECK FOR EOF
        JRZ     ..RET   ;NO SO NORMAL RETURN
        BIT     0,0(Y)  ;SEE IF OVERFLOW PENDING
        JRNZ    ..RET   ;YES SO NORMAL RETURN
        STC
..RET:  POP     B
        POP     D
        RET



;
;       HANDLE OVERFLOW
;
..UGH:  LXI     H,LINBUF#




        MVI     C,20
        MOV     B,C
..U1:   MVI     M,' '   ;CLEAR OFFSET IN LINE
        INX     H
        DJNZ    ..U1
        MOV     E,1(Y)  ;LOAD OVERFLOW PTR
        MOV     D,2(Y)
        MOV     B,3(Y)  ;LOAD OVERFLOW COUNT
..U2:   LDAX    D
        MOV     M,A     ;MOVE IN OVERFLOW
        INX     D
        INX     H
        INR     C
        DJNZ    ..U2
        MOV     A,C
        JMPR    ..GO    ;NOW HANDLE AS NORMAL LINE



;
;       HANDLE LONG LINE
;
..LNG:  MVI     B,0
..LL1:  MOV     A,M
        CPI     ' '     ;BACK SCAN
        JRZ     ..OUT
        CPI     ','
        JRZ     ..OUT
..LL2:  DCX     H
        INR     B
        DCR     C
        JMPR    ..LL1
..OUT:  LDA     HLEN#
        DCR     A
        DCR     A
        CMP     C
        JRC     ..LL2   ;STILL TOO LONG
        INX     H
        INR     C
        DCR     B
        MOV     1(Y),L
        MOV     2(Y),H
        MOV     3(Y),B
        SET     0,0(Y)  ;SET OVERFLOW FLAG
        JMPR    ..MVE
```

```
;
;       GET NEXT FILE LINE



;
GETLIN: MVI     C,0
        LXI     H,LINBUF#
..LP:   FGET    INP,EOF
        CPI     1AH
        JRZ     EOF     ;END OF FILE FOUND
        CPI     13
        JZ      CR      ;END OF LINE
        CPI     10



        JRZ     ..LP    ;SKIP LINE FEEDS
        CPI     9
        JRZ     ..TB    ;EXPAND TABS
..TB2:  MOV     M,A     ;SAVE IN BUFFER
        CPI     8
        JRZ     ..OK    ;BS OK
        CPI     ' '-1


        JRNC    ..OK
        MVI     M,1FH   ;"UNPRINTABLE" CHARACTER
..OK:   INX     H
        INR     C
        MVI     A,0FFH
        CMP     C
        JZ      CR      ;FORCE END OF LINE
        JMPR    ..LP    ;LOOP TILL END OF LINE


;
;       HANDLE (EXPAND) TABS
;



..TB:   MOV     A,C
        ADI     8
        ANI     0F8H
        SUB     C
        MOV     B,A
        MVI     A,' '
..TB1:  MVI     M,' '   ;SET BLANKS


        INX     H
        INR     C       ;COUNT
        DJNZ    ..TB1
        JMPR    ..LP    ;AND KEEP GOING


;
;       END OF LINE
;
CR:     MOV     A,C     ;LOAD COUNT
        RET
;



;       END OF FILE
;
EOF:    SET     1,0(Y)  ;SET EOF FLAG
        JMPR    CR
```

(Continued on next page)

CP/M Print Utility

(Listing continued, text begins on page 36)

```
;
;       SCAN PAGE MATRIX AND SET CR/LF
;
SCAN:   LHLD    .FREE.          ;POINT TO END OF LINE
        LDED    LLEN#   ;LOAD LINE LENGTH
        DAD     D
        DCX     H       ;HL POINTS TO END OF LINE
        LDA     LINES#
        MOV     B,A     ;LINE COUNT
..LP:   CALL    REVRS   ;DO REVERSE SCAN
        DAD     D
        DJNZ    ..LP
        RET
;
;       REVERSE SCAN FOR NON BLANK
;
REVRS:  PUSH    H
        PUSH    B
        LBCD    LLEN#   ;LINE LENGTH
        MVI     A,' '
..LP:   CCD
        JNO     ..GO
        JRZ     ..LP
..GO:   INX     H
        INX     H
        MVI     M,13    ;SET CR
        INX     H
        MVI     M,10    ;SET LF
        POP     B
        POP     H
        RET
;
        .END
```

Listing 6

```
Listing of DPRWS.ASM      March 18, 1982

        .TITLE  "DOUBLE COLUMN PRINT - WORKING STORAGE"
        .SBTTL  "VERSION 2.0 - MARCH 1982 - DD SYSTEMS"
        .IDENT  DPRWS
;
        .INSERT MACF1
;
;       THE MODULE CONTAINS THE PRIMARY WORKING STORAGE FOR THE
                PROGRAM
;
;       SYSTEM EQUATES
;
BUF1    ==      80H     ;CP/M DEFAULT BUFFER
;
;       DUMMY FCB'S
;
        FCB     INP,,,,BUF1
        FCB     SRCH,,,,BUF2
;
STKPTR::.BLKW   1       ;CURRENT STACK ENTRY
STKCNT::.BLKB   1       ;STACK COUNT
STACK:: .BLKB   768     ;64 ENTRY STACK
;
LINSX:: .BLKB   4       ;LINE CONTROL POINTERS AND SWITCH
LINBUF::.BLKB   255     ;LINE INPUT BUFFER
;
PGCNT:: .BLKB   1       ;PAGE COUNTER
;
BUF2::  .BLKB   128     ;SEARCH FILE BUFFER
;
        .END
```

End Listing

Cross-Reference Generator in C

A Program Conversion Aid

by Jeffrey L. Taylor

Jeffrey L. Taylor, The Toolsmith, 139 G Street, Box 151, Davis, CA 95616.

Cross-reference listings are of little use to the person who originally wrote the program, so most such programs just gather dust on the back shelf. But those of us who don't own Apples or TRS-80s get much of our software by converting programs written for some other computer. Cross-reference listings save hours of debugging on conversion efforts. When I started to convert Ron Cain's Small-C compiler to the LSI-11, I hauled XREF, a cross-reference program I had written years before, off the shelf where it had been resting. Ron's different coding style revealed a few additional errors. However, the biggest problem was the symbol table, for the compiler would not fit in memory. CREF, the program described in this article, is a test bed for a different symbol table implementation that is more compact. Eventually, the more compact symbol table implementation will get folded back into XREF.

CREF is written for Whitesmith's C compiler and Portable C library. The language is compatible with Unix Version 7, but the library functions are not. For those with different compilers and libraries, Figure 2 shows a cross-reference listing of the library routines. CREF accepts a list of file names from the command line; if there are no file names, input is taken from the standard input. The cross-reference listing is printed on the standard output. The listing consists of all file names in "#include" statements, and all function, variable, and macro names in alphabetical order. Under each name is a list of files it appears in. To the right of each file name is a list of occurrences by line number.

The three-level hierarchy in the listing is reflected in the symbol table organization as shown in Figure 1. At the top is the name table, next a file table for each name, and at the bottom a list of line numbers for each file and name. While each name entry is unique, there is a file entry for each name in the file. This is where the symbol table was reorganized; how it was changed is explained later.

Binary trees are used for the name- and file-search trees. Each entry contains text for a symbol and pointers, possibly NULL, to the left and right decendents. A binary tree grows from the "root" down, with new entries added at the tips or "leaves." The function lookup finds the entry for a symbol, or if it is not present, where it should go. To search a binary tree, start at the root and compare the symbol against that stored in the entry. If it matches, stop, this is it. If it is before the entry in the alphabet, move to the left branch. Otherwise, it is after the entry and move to the right branch. Repeat until it is found or there is no branch; i.e., the NULL pointer is encountered. If the name is not found, that last NULL branch is where it should be added. The tree is always kept in alphabetical order. The function "tree_walk" traverses a binary tree in what is known as in-order or symmetric order. It calls itself to visit all entries in the left subtree, then the original entry, and then all entries in the right subtree. Short, simple code, it uses lots of stack space.

The list of line numbers is kept in a linked list. Each entry contains a line number and a pointer or link to the next line number. The pointer for the last line number is NULL. If a name occurs twice in the same line, then the line number is only recorded once. As each line number is added to the list, it is inserted at the top. Any duplicates are on top, so each line number is compared to the top of the list before adding. The function reverse is called by "list_file" to flip the order just before the list is printed.

This symbol table structure as given works fine. However, for large programs there is considerable redundancy. There is a file entry for every name in it. If the text for that file name is stored separate form the rest of the entry, all entries for that file can use the same text. A point to the text is stored in each entry. To keep the code simple, this separate text entry is used for all trees. A tree rooted at "file_root" keeps track of the text for all file names to avoid duplicates. Name entries are looked up and added in a straightforward manner. File entries are looked up similarly, but adding a new entry is more complicated. The entry is added as usual, but "file_root" is searched and possibly added to for the text of the file name.

CREF is intended for cross-referencing C programs. By changing the reserved word list and the code that ignores comments, it can be adapted to most other languages. I don't currently have any programs too large for CREF, but sooner or later they will come along. When that happens, I must learn about some new data structure to store part of the symbol table on disk. It looks like a good way to learn about B-trees. ▶▶J

(Listing begins on page 51)

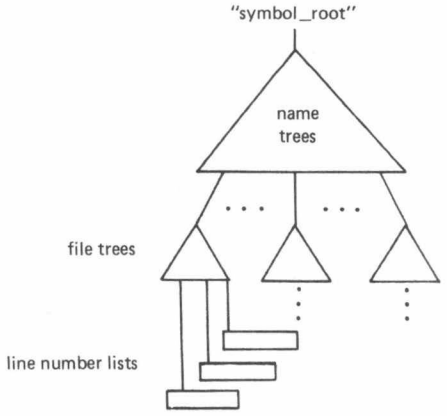


Figure 1 – Symbol Table Organization

```
alloc
    cref.c   61   64
    linked.c   26   28
    string.c   55   61
btoi
    cref.c   231   258
fclose
    cref.c   316   326
fopen
    cref.c   316   322
getlin
    cref.c   138   142
itob
    cref.c   280   292
putlin
    cref.c   272
stdin
    cref.c   316   322   326
write
    diagn.c   9   14   16   17
```

Figure 2 – Library Cross Reference

Cross-Reference Generator

(Text begins on page 50)

```
cref.c       1: #include <std.h>
cref.c       2: #include <local.h>
cref.c       3: /*******************************
cref.c       4: *  C cross reference utility  *
cref.c       5: *******************************/
cref.c       6:
cref.c       7: /* (c) 1982 by The Toolsmith */
cref.c       8:
cref.c       9: typedef short boolean;
cref.c      10:
cref.c      11: #include "lexcmp.c"
lexcmp.c     1:
lexcmp.c     2: /* copyright (c) 1982 by The Toolsmith */
lexcmp.c     3:
lexcmp.c     4: /* lexcmp - lexical comparison, -1 ("a" < "b"), 0 ("a" == "b"), or +1 */
lexcmp.c     5: int lexcmp(aa, bb, nn)
lexcmp.c     6:   char *aa, *bb;
lexcmp.c     7:   unsigned nn;
lexcmp.c     8:   {
lexcmp.c     9:   register char *a, *b;
lexcmp.c    10:   register unsigned n;
lexcmp.c    11:
lexcmp.c    12:   for(a = aa, b = bb, n = nn; ;a++, b++) {
lexcmp.c    13:     if(tolower(*a) != tolower(*b))
lexcmp.c    14:       return tolower(*a) - tolower(*b); /* unequal */
lexcmp.c    15:     if(*a == '\0' || --n == 0)
lexcmp.c    16:       break;
lexcmp.c    17:     }
lexcmp.c    18:   for(a = aa, b = bb, n = nn; ; a++, b++) {
lexcmp.c    19:     if(*a != *b)
lexcmp.c    20:       return *a - *b;   /* unequal */
lexcmp.c    21:     if(*a == '\0' || --n == 0)
lexcmp.c    22:       break;
lexcmp.c    23:     }
lexcmp.c    24:   return 0;      /* equal */
lexcmp.c    25:   }
cref.c      12: #include "diagn.c"
diagn.c      1:
diagn.c      2: /* (c) copyright 1982 by The Toolsmith */
diagn.c      3:
diagn.c      4: /* diagnostic - spit out error message */
diagn.c      5: diagnostic(fatal, args)
diagn.c      6:   boolean fatal;
diagn.c      7:   char *args;
diagn.c      8:   {
diagn.c      9:   extern unsigned lenstr(), write();

diagn.c     10:   extern exit();
diagn.c     11:   register char **a;
diagn.c     12:
diagn.c     13:   if(fatal)
diagn.c     14:     write(STDERR, "PANIC!!! ", 9);
diagn.c     15:   for(a = &args; *a != NULL; ++a)
diagn.c     16:     write(STDERR, *a, lenstr(*a));
diagn.c     17:   write(STDERR, "\n", 1);
diagn.c     18:   if(fatal)
diagn.c     19:     exit(NO);   /* error exit to OS */
diagn.c     20:   }
cref.c      13: #include "esc.c"
esc.c        1:
esc.c        2: /* (c) copyright 1982 by The Toolsmith */
esc.c        3:
esc.c        4: /*  esc - map string into escaped character if appropriate  */
esc.c        5: char esc(s)
esc.c        6:   register char **s;
esc.c        7:   {
esc.c        8:   extern unsigned scnstr();
esc.c        9:   static char real[] = "\b\t\v\f\n\r\(\!\)\^";
esc.c       10:   static char symb[] = "btvfnr(!)^";
esc.c       11:
esc.c       12:   if(**s != '\\')
esc.c       13:     return **s;
esc.c       14:   if(*(*s+1) == '\0')   /* not special at end */
esc.c       15:     return '\\';
esc.c       16:   ++*s;
esc.c       17:   real[sizeof(real)-1] = **s;
esc.c       18:   return real[scnstr(symb, **s)];
esc.c       19:   }
cref.c      14: #include "linked.c"
linked.c     1:
linked.c     2: /* (c) copyright 1982 by The Toolsmith */
linked.c     3:
linked.c     4: struct linked_list {
linked.c     5:   struct linked_list *link;
linked.c     6:   };
linked.c     7:
linked.c     8: /* reverse - invert order of a linked list */
linked.c     9: struct linked_list *reverse(this)
linked.c    10:   register struct linked_list *this;
linked.c    11:   {
linked.c    12:   register struct linked_list *next, *previous;
linked.c    13:
linked.c    14:   for(previous = NULL; this != NULL; previous = this, this = next) {
linked.c    15:     next = this->link;
linked.c    16:     this->link = previous;
linked.c    17:     }
linked.c    18:   return previous;
linked.c    19:   }
linked.c    20:
linked.c    21: /* insert - add an "n" byte element to front of "list" */
linked.c    22: struct linked_list *insert(n, list)
linked.c    23:   unsigned n;
linked.c    24:   struct linked_list *list;

linked.c    25:   {
linked.c    26:   extern char *alloc();
linked.c    27:
linked.c    28:   return (struct linked_list *) alloc(n, list);
linked.c    29:   }
cref.c      15: #include "string.c"
string.c     1:
string.c     2: /* (c) copyright 1982 by The Toolsmith */
string.c     3:
string.c     4: /* cmpstr - compare strings */
string.c     5: boolean cmpstr(a, b)
string.c     6:   register char *a, *b;
string.c     7:   {
string.c     8:   for( ; *a == *b++; ++a)
string.c     9:     if(*a == '\0')
string.c    10:       return YES;
string.c    11:   return NO;
string.c    12:   }
string.c    13:
string.c    14: /* cpybuf - copy buffer "s" to "d" for "n" bytes */
string.c    15: unsigned cpybuf(d, s, n)
string.c    16:   register char *d, *s;
string.c    17:   unsigned n;
string.c    18:   {
string.c    19:   register unsigned i;
string.c    20:
string.c    21:   for(i = n; i != 0; --i)
string.c    22:     *d++ = *s++;
string.c    23:   return n;
string.c    24:   }
string.c    25:
string.c    26: /* instr - find first occurrence in string of member of set */
string.c    27: unsigned instr(p, s)
string.c    28:   register char *p, *s;
string.c    29:   {
string.c    30:   register char *c;

string.c    31:
string.c    32:   for( ; *s; s++)
string.c    33:     for(c = p; *c; c++)
string.c    34:       if(*c == *s)
string.c    35:         goto out;
string.c    36: out:
string.c    37:   return c - p;
string.c    38:   }
string.c    39:
string.c    40: /* lenstr - returns length of string "s" */
string.c    41: unsigned lenstr(s)
string.c    42:   register char *s;
string.c    43:   {
string.c    44:   register unsigned n;
string.c    45:
string.c    46:   for(n = 0; *s++ != '\0'; ++n)
string.c    47:     ;
string.c    48:   return n;
string.c    49:   }
string.c    50:
string.c    51: /* newcpy - copy a string into newly allocated space */
string.c    52: char *newcpy(s)
string.c    53:   char *s;
string.c    54:   {
string.c    55:   extern char *alloc();
string.c    56:   extern unsigned lenstr();
string.c    57:   register char *t;
string.c    58:   register unsigned n;
string.c    59:
string.c    60:   n = lenstr(s) + 1;    /* include terminating NUL */
string.c    61:   t = alloc(n);
string.c    62:   cpybuf(t, s, n);
string.c    63:   return t;
string.c    64:   }
string.c    65:
```

(Continued on next page)

Cross-Reference Generator (Listing continued, text begins on page 50)

```
string.c    66: /* scnstr - scan "s" for first occurrence of "c" */
string.c    67: unsigned scnstr(s, c)
string.c    68:   register char *s, c;
string.c    69:   {
string.c    70:   register char *t;
string.c    71:
string.c    72:   for(t = s; *t && *t != c; ++t)
string.c    73:     ;
string.c    74:   return t - s;
string.c    75:   }
cref.c      16:
cref.c      17: struct instance {
cref.c      18:   struct instance *link;
cref.c      19:   int line;
cref.c      20:   };
cref.c      21: union ptr {
cref.c      22:   struct node *files;
cref.c      23:   struct instance *lines;
cref.c      24:   };
cref.c      25: struct node {
cref.c      26:   struct node *right, *left;
cref.c      27:   union ptr p;
cref.c      28:   char *name;
cref.c      29:   };
cref.c      30:
cref.c      31: struct node *symbol_root = NULL;
cref.c      32: struct node *file_root = NULL;
cref.c      33: #define ID 'a'  /* identifier */
cref.c      34: #define INTEGER '0'     /* integer */
cref.c      35:
cref.c      36: /**********************************
cref.c      37: *  Symbol Table handling routines  *
cref.c      38: **********************************/
cref.c      39:
cref.c      40: /* lookup - install name at or below root */
cref.c      41: struct node **lookup(root, name)
cref.c      42:   register struct node **root;
cref.c      43:   register char *name;
cref.c      44:   {
cref.c      45:   register int cond;
cref.c      46:
cref.c      47:   if(*root != NULL) {
cref.c      48:     if((cond = lexcmp(name, (*root)->name, 0)) < 0)
cref.c      49:       root = lookup(&(*root)->left, name);
cref.c      50:     else if(cond > 0)
cref.c      51:       root = lookup(&(*root)->right, name);
cref.c      52:     }
cref.c      53:   return root;
cref.c      54:   }
cref.c      55:
cref.c      56: /* add - insert entry for "name" in tree at "root" */
cref.c      57: add(root, name)
cref.c      58:   struct node **root;
cref.c      59:   char *name;
cref.c      60:   {
cref.c      61:   extern char *alloc();
cref.c      62:   register struct node *r;
cref.c      63:
cref.c      64:   r = (struct node *) alloc(sizeof(struct node), NULL);
cref.c      65:   r->left = r->p.lines = NULL;
cref.c      66:   r->name = name;
cref.c      67:   *root = r;
cref.c      68:   }
cref.c      69:
cref.c      70: /* tree_walk - walk a binary tree, doing ftn to each node */
cref.c      71: tree_walk(root, ftn)
cref.c      72:   register struct node *root;
cref.c      73:   register (*ftn)();
cref.c      74:   {
cref.c      75:   if(root != NULL) {
cref.c      76:     tree_walk(root->left, ftn);
cref.c      77:     (*ftn)(root);
cref.c      78:     tree_walk(root->right, ftn);
cref.c      79:     }
cref.c      80:   }
cref.c      81:
cref.c      82: /* use - log an occurrence of "name" in "file" at "line" */
cref.c      83: use(name, file, line)
cref.c      84:   char *name, *file;
cref.c      85:   int line;
cref.c      86:   {
cref.c      87:   register struct instance **it;
cref.c      88:   register struct node **ft, **nt;
cref.c      89:
cref.c      90:   if(*(nt = lookup(&symbol_root, name)) == NULL)
cref.c      91:     add(nt, newcpy(name));
cref.c      92:   if(*(nt = lookup(&((*nt)->p.files), file)) == NULL) {
cref.c      93:     if(*(ft = lookup(&file_root, file)) == NULL)
cref.c      94:       add(ft, newcpy(file));
cref.c      95:     add(nt, (*ft)->name);
cref.c      96:     }
cref.c      97:   it = &((*nt)->p.lines);
cref.c      98:   if(*it == NULL || (*it)->line != line) {
cref.c      99:     *it = (struct instance *) insert(sizeof(struct instance)
cref.c     100:     (*it)->line = line;
cref.c     101:     }
cref.c     102:   }
cref.c     103:
cref.c     104: /* get_name - extract file name from line */
cref.c     105: unsigned get_name(line, file)
cref.c     106:   register char *line;
cref.c     107:   char *file;
cref.c     108:   {
cref.c     109:   register unsigned n;
cref.c     110:   register char *delim;
cref.c     111:
cref.c     112:   while(*line == ' ' || *line == '\t')
cref.c     113:     ++line;
cref.c     114:   if(*line == '\n')
cref.c     115:     n = lenstr(file);
cref.c     116:   else {
cref.c     117:     if(*line == '"') {
cref.c     118:       delim = "\"\n";
cref.c     119:       ++line;
cref.c     120:       }
cref.c     121:     else if(*line == '<') {
cref.c     122:       delim = ">\n";
cref.c     123:       ++line;
cref.c     124:       }
cref.c     125:     else
cref.c     126:       delim = " \t\n";
cref.c     127:     n = cpybuf(file, line, instr(line, delim));
cref.c     128:     file[n] = '\0';
cref.c     129:     }
cref.c     130:   return n;
cref.c     131:   }
cref.c     132:
cref.c     133: unsigned line_count = 0;
cref.c     134:
cref.c     135: /* new_line - return pointer to the next line */
cref.c     136: char *new_line()
cref.c     137:   {
cref.c     138:   extern unsigned getlin();
cref.c     139:   static char line[MAXLINE+1];
cref.c     140:
cref.c     141:   ++line_count;
cref.c     142:   line[getlin(line, MAXLINE)] = '\0';
, *it);
cref.c     143:   return line;
cref.c     144:   }
cref.c     145:
cref.c     146:
cref.c     147: /* white_space - tests for blanks, tabs and comments */
cref.c     148: boolean white_space(s)
cref.c     149:   register char **s;
cref.c     150:   {
cref.c     151:   if(**s == ' ' || **s == '\t')
cref.c     152:     return YES;
cref.c     153:   if(**s == '/' && *(*s+1) == '*') {    /* comment */
cref.c     154:     while(*++*s != '/')
cref.c     155:       while(*++*s != '*')
cref.c     156:         if(**s == '\0')
cref.c     157:           if(*(*s = new_line()))
cref.c     158:             --*s;
cref.c     159:           else {
cref.c     160:             diagnostic(NO, "unexpected EOF", NULL);
cref.c     161:             return NO;
cref.c     162:             }
cref.c     163:     return YES;
cref.c     164:     }
cref.c     165:   return NO;
cref.c     166:   }
cref.c     167:
cref.c     168: /* get_token - strip leading token from s */
cref.c     169: char get_token(s, t)
cref.c     170:   register char **s, *t;
cref.c     171:   {
cref.c     172:   register char class;
cref.c     173:   #define ishex(c) ('0'<=(c)&&(c)<='9'||'a'<=(c)&&(c)<='f'|
                                                    |'A'<=(c)&&(c)<='F')
cref.c     174:
cref.c     175:   while(white_space(s))
cref.c     176:     ++*s;
cref.c     177:   if(isalpha(**s) || **s == '_') {      /* identifier */
cref.c     178:     class = ID;
cref.c     179:     do
cref.c     180:       *(t++) = *((*s)++);
cref.c     181:     while(isdigit(**s) || isalpha(**s) || **s == '_');
cref.c     182:     }
```

```
cref.c    183:   else if(**s == '"' || **s == '\'') {  /* string or literal */
cref.c    184:     class = **s;
cref.c    185:     do {
cref.c    186:       esc(s);
cref.c    187:       ++*s;
cref.c    188:       if(**s == '\0')
cref.c    189:         if(*(*s = new_line()) == '\0')
cref.c    190:           goto out;
cref.c    191:       } while(**s != class);
cref.c    192:     ++*s;
cref.c    193:     }
cref.c    194:   else if(isdigit(**s)) {
cref.c    195:     do {
cref.c    196:       class = *++*s;
cref.c    197:       class = tolower(class);
cref.c    198:       } while(ishex(class) || class == 'x' || class == 'l' |
                                                    | class == '.');
cref.c    199:     class = INTEGER;
cref.c    200:     }

cref.c    201:   else {
cref.c    202:     class = **s;
cref.c    203:     ++*s;
cref.c    204:     }
cref.c    205: out:
cref.c    206:   *t = '\0';
cref.c    207:   return class;
cref.c    208:   }
cref.c    209:
cref.c    210: /* keyword - is "s" a reserved word in C */
cref.c    211: boolean keyword(s)
cref.c    212:   register char *s;
cref.c    213:   {
cref.c    214:   register char **k;
cref.c    215:   register int c;
cref.c    216:   static char *reserved[] = {"auto", "break", "case", "char", "continue",
cref.c    217:     "default", "do", "double", "else", "extern", "float", "for", "goto",
                                                                        "if",
cref.c    218:     "int", "long", "register", "return", "short", "sizeof", "static",
cref.c    219:     "struct", "switch", "typedef", "union", "unsigned", "while", NULL};
cref.c    220:
cref.c    221:   for(k = reserved; *k != NULL; ++k)    /* reserved must be sorted */
cref.c    222:     if((c = lexcmp(s, *k)) <= 0)
cref.c    223:       break;
cref.c    224:   return c == 0;        /* equality */
cref.c    225:   }
cref.c    226:
cref.c    227: /* xref - cross reference */
cref.c    228: xref(file)
cref.c    229:   register char *file;
cref.c    230:   {
cref.c    231:   extern unsigned btoi();
cref.c    232:   register char c;
cref.c    233:   char *s, token[MAXLINE+1];
cref.c    234:
cref.c    235:   line_count = 0;
cref.c    236:   while(*(s = new_line())) {
cref.c    237:     if((c = get_token(&s, token)) != '#')
cref.c    238:       while(c != '\n') {
cref.c    239:         if(c == ID && !keyword(token))
cref.c    240:           use(token, file, line_count);
cref.c    241:         c = get_token(&s, token);
cref.c    242:         }
cref.c    243:       else if(get_token(&s, token) == ID) {
cref.c    244:         if(cmpstr(token, "include")) {
cref.c    245:           get_name(s, token);
cref.c    246:           use(token, file, line_count);

cref.c    247:           }
cref.c    248:         else if(cmpstr(token, "define")) {
cref.c    249:           get_token(&s, token);
cref.c    250:           use(token, file, line_count);
cref.c    251:           }
cref.c    252:         else if(cmpstr(token, "ifdef") || cmpstr(token, "ifndef")) {
cref.c    253:           get_token(&s, token);
cref.c    254:           use(token, file, line_count);
cref.c    255:           }
cref.c    256:         else if(cmpstr(token, "line")) {
cref.c    257:           if(get_token(&s, token) == INTEGER)
cref.c    258:             btoi(token, MAXLINE, &line_count, 10);
cref.c    259:           else
cref.c    260:             diagnostic(NO, "#line ", token, NULL);
cref.c    261:           }
cref.c    262:         else
cref.c    263:           diagnostic(NO, "#", token, *s, NULL);
cref.c    264:         }
cref.c    265:       }
cref.c    266:   }
cref.c    267:
cref.c    268: /* puts - output s string to STDOUT */
cref.c    269: unsigned puts(s)
cref.c    270:   char *s;
cref.c    271:   {
cref.c    272:   return putlin(s, lenstr(s));
cref.c    273:   }
```

(Continued on next page)

```
cref.c    274:
cref.c    275:
cref.c    276: /* list_file - print lines within a file */
cref.c    277: list_file(ft)
cref.c    278:   register struct node *ft;
cref.c    279:   {
cref.c    280:   extern unsigned itob();
cref.c    281:   register unsigned b;
cref.c    282:   register struct instance *it;
cref.c    283:   char buf[5];
cref.c    284:
cref.c    285:   b = puts("    ");

cref.c    286:   b += puts(ft->name);
cref.c    287:   /* print line numbers */
cref.c    288:   for(it = reverse(ft->p.lines); it != NULL; it = it->link) {
cref.c    289:     if(b == 0)
cref.c    290:       b = puts("       ");      /* this and 2nd to last line
                                                          must agree */
cref.c    291:     b += puts("   ");
cref.c    292:     buf[itob(buf, it->line, 0)] = '\0';

cref.c    293:     b += puts(buf);
cref.c    294:     if(b > HARD_WIDTH-10) {
cref.c    295:       puts("\n");
cref.c    296:       b = 0;
cref.c    297:       }
cref.c    298:     }
cref.c    299:   if(b > 6)     /* non-blank line */
cref.c    300:     puts("\n");
cref.c    301:   }
cref.c    302:

cref.c    303: /* print_xref - dump cross reference table to stdout */
cref.c    304: print_xref(nt)
cref.c    305:   struct node *nt;
cref.c    306:   {
cref.c    307:   puts(nt->name);
cref.c    308:   puts("\n");
cref.c    309:   tree_walk(nt->p.files, &list_file);
cref.c    310:   }
cref.c    311:

cref.c    312: boolean main(ac, av)
cref.c    313:   register int ac;
cref.c    314:   register char **av;
cref.c    315:   {

cref.c    316:   extern FIO *fclose(), *fopen(), stdin;
cref.c    317:
cref.c    318:   if(ac <= 1)
cref.c    319:     xref("<stdin>");
cref.c    320:   else

cref.c    321:     while(--ac > 0)
cref.c    322:       if(fopen(&stdin, *++av, READ) == NULL)
cref.c    323:         diagnostic(NO, "can't open ", *av, NULL);
cref.c    324:       else {
cref.c    325:         xref(*av);
cref.c    326:         fclose(&stdin);
cref.c    327:         }

cref.c    328:   tree_walk(symbol_root, &print_xref);
cref.c    329:   return YES;
cref.c    330:   }
```

End Listing

Letters *(Continued from page 6)*

CO Debugger (ZDT?), Version 00.16. Its capabilities exceed those of DDT in many ways, so I suspect it is better than ZSID in all but its lack of symbol handling.

By using these packages, I encounter the following problems:

1) The definitions module and the COMMON area must be INCLUDEd with each module assembly, significantly increasing the length of time it takes to assemble, and increasing the size of the cross-reference listing produced by each module assembly. As well, any change affecting the relative positions of symbols in the COMMON area necessitates the reassembly of all 23 modules.
2) No program-wide cross-reference listing is possible. (At an earlier stage of program development, all the modules were coded absolute, and were INCLUDE files in a single module which was assembled absolute, producing a .HEX file. This made it possible to generate a cross-reference listing of all symbol usages in all modules. As the program size increased, this became no longer possible, due to the Assembler running out of memory.)
3) The debugger, although excellent in all other respects, lacks the ability to use symbol names and values from a .SYM file.

Problem (1) is created to some extent by the linker's inability to perform more than simple addition and subtraction of constants to externals. More significant, however, is the assembler's inability to handle 8-bit and 3-bit externals. For example, it would be desirable to assemble the following instruction:

```
BIT FLAG,(IX+LOCN)
```

where both FLAG and LOCN are externals. The value of FLAG ultimately should appear in object code as bits 5,4,3 of the 4th byte of the instruction, and the value of LOCN should appear as the 3rd byte of the instruction. (Microsoft's Macro-80 and Link-80 3.43 seem to have an undocumented 8-bit capability, but Macro-80 always assembles the 3-bit reference as absolute zeroes, with no error message.)

Problem (2) is given only a very weak solution by the Microsoft Link-80's /L option, applied to a .LIB file composed of the 23 .REL files. The problem with the /L listing is that the symbols are ordered as they are referenced by modules, and not alphabetically. Also, multiple references by a module to an external symbol are not enumerated. What is really needed is a listing of each global symbol, in alphabetical order, giving its defining module and relative value, together with the name of each module referencing that symbol, and the relative locations within that module that contain the references.

As for problem (3), I know of no symbolic debugger that offers the ability to edit instructions (displaying each instruction before asking for input under the A command), general arithmetic expression evaluation, and specification of a relocation constant specifying the base address of a module, so that relative addresses can be used in commands. (The CROMEMCO Debugger does all but handle symbols.)

In summary, I seek the "ideal" assembly-link-debug package for Z80 development under CP/M. It should have the following features:

a. Zilog/Mostek Z80 mnemonics, of course.
b. Internal symbol names of at least 8 characters.
c. External symbol names of at least 7 characters.
d. 16-bit, 8-bit, and 3-bit externals.
e. IF-THEN-ELSE conditional.
f. Macros with string operators.
g. STRUCT and GLOBAL features of the CROMEMCO Assembler.
h. The ability to link 123 global symbols and 32K of object code under a 53K transient area.
i. Local cross-reference within modules for all internal and external symbols, listing lower-case symbol names in lower-case.
j. Global cross-reference among all modules for all external symbols, listing their values and locations of reference.

Should readers attempt to partially beg the question by suggesting a high-level language be used instead of assembly language, I ask that they be sure that it can generate the optimized high speed, small size, interrupt-driven code demanded by a Z80 real-time control application. The language should be able to be compiled by any CP/M system, run standalone without CP/M, be self-documenting and very readable, support data structures and string operations, and be available for purchase without restrictions regarding the hardware it is used with, or licensing fees regarding the distribution of programs created under it.

I'm even willing to consider operating systems other than CP/M, provided they support its extensive customization capabilities; i.e., run on the same custom computer, and read the same custom-formatted disks (at least once).

Hopefully the publication of this letter and any responses will help others like myself who find that their system programming tools fall short of the tasks required of them. if you have a favorite assembler, linker, debugger, language, or system that meets the above requirements, please be so kind as to share it with us.

Yours for better software,
Wayne Farmer
R.A. Gray, Inc.
9181 Chesapeake Drive
San Diego, CA 92123

Centrifugal Forth

Dear Marlin,

The article on "Runic" in *DDJ* #65 is one of many fulfillments of my rueful prophecy in *DDJ* #51-52 that the limitations of Forth would inspire creation of incompatible look-alikes. Others are Ron Loeliger's ZIP (in his *Threaded Interpretive Languages*, BYTE Books, 1981), Samurai Software's RPL (March 1982 *BYTE*), and Motalygo's PS (October 1981 *BYTE*). It's likely that in the increasing number of versions of "standard" Forth there will be minor tinkering enhancements that may cause subtle incompatibilities. Every commercial version tries to be "better" than its rivals, and this often involves optimizations to "fit" one of the major hardware systems or innovations to appeal to special user groups. I wonder if any "virtual machine" concept is stong and flexible enough to withstand these centrifugal forces!

Sincerely,
H. T. Gordon
College of Natural Resources
University of California
Berkeley, CA 94720

BOOK REVIEWS

Elementary Pascal
by Henry Ledgard & Andrew Singer
Vintage Books
266 pages, $12.95
Circle No. 379 on Reader Service Card
Reviewed by D. C. Shoemaker

Judging from the number of books appearing on the market these days, Pascal is still the language with the fastest-growing following. It seems that at least one or two introductory texts reach my bookseller's shelves each month, and no sign of letting up. Naturally, some are better than others, and some are more advanced than others, and some are more specialized than others. *Elementary Pascal* is somewhat difficult to classify, so let me tell you about it, and let you decide where it fits.

Henry Ledgard is no stranger to computer programmers, or to users of Pascal. Responsible in part or in total for the "Programming Proverbs" series, including *Pascal with Style*, he is one of the better writers who take it upon themselves to get the novice programmer off on the right foot, and perform a valuable service thereby. His latest book is no exception.

Ledgard chose to combine an introductory-level discussion of Pascal with a whimsical touch of humor, nostalgia and mystery by setting his work in the late Victorian English environs so well known to fans of Conan Doyle's Sherlock Holmes. His medium is a newly discovered trunkful of manuscripts by Holmes' biographer, Dr. Watson, with Ledgard and Singer the modern-day "editors." When I first picked up the book, I was rather skeptical that this approach would work at all, but work it does, and well.

It develops that Holmes was aware of the work of Charles Babbage on the Analytical Engine, and that he has access to a much-improved version. It turns out that this improved version speaks Pascal, and Holmes, with his logical mind, is an ace programmer. Much of the book deals with Holmes' introduction of Watson to the Engine, and to Pascal. In passing, Holmes is able to develop the concept of logical structure and problem-solving techniques, with the eternal Watson as the student and foil.

A large number of people other than the authors worked very hard to insure the success of this approach, and they are given full credit in an interesting postscript at the conclusion of the book. A Holmes fan myself, I must admit that I was at least as interested in the authors' approach and how they did what they did, as I was in the results themselves. They manage to capture the flavor of Sir Arthur's characters quite well, partly by the time-honored approach of lifting short passages from other Holmes tales and recombining them into new stories. They are not the first to do this, but they have done a better job than most, partly because they don't push too hard to make a full-length story.

The book consists of fourteen chapters, each of which develops a lesson on the use or application of some feature of Pascal in the solution of a mystery. Chapters are divided into two parts, the first being the discussion between Holmes and Watson about the mystery itself, stressing the logic of the approach to a solution, and the second being a more detailed examination of the program techniques used. Topics range from a discussion of algorithms to structure to punctuation to statements. Arrays, procedures and functions, input and output, record structures and subprograms are all covered by means of examples. In short, all the major aspects of Pascal are dealt with by the end of the book, and an appendix called "Pascal at a Glance" summarizes the language. The version of Pascal used is almost identical to X3J9/81-003, the draft, proposed ISO Standard Pascal, and I would judge that most Pascal users will find the examples close enough to their particular versions to be useful.

Intended as an introduction, to be read and used in conjunction with the manual for a specific implementation of the language, *Elementary Pascal* makes a very effective first book on programming, one that will help keep the beginner on the path to good programming practices and help prevent him or her from falling into the most common errors. Taken in this respect, the book is excellent, and even more experienced programmers, especially those not familiar with Pascal, can benefit from it. The authors say they will be following up with similar books on other languages, and the BASIC version is already in print. It appears identical to the Pascal version, with the only discernable differences due to variations in the languages themselves. A unique approach, and a unique book, highly recommended.

Pascal Programs for Scientists and Engineers
by Alan R. Miller
Sybex
374 pages, $16.95
Circle No. 389 on Reader Service Card
Reviewed by William G. Hutchison, Jr.

Do you know some Pascal and some math, but have difficulty putting the two together? If so, Alan Miller's new book *Pascal Programs for Scientists and Engineers* will help. Miller is a Professor of Metallurgy at the New Mexico Institute of Mining and Technology. He has also written several books and numerous articles on microcomputers and programming.

Miller's practical orientation is evident throughout the book. Unlike some authors, he doesn't begin with an axiomatic definition of Pascal or with abstract principles of program design. He starts with examples of "correct" Pascal programs which produce incorrect answers with some current small-computer Pascal compilers. The examples mostly relate to floating-point exceptions and trig functions, which the example compilers do not handle correctly.

The remainder of the book covers the usual material found in most introductory numerical analysis books: elementary statistics, matrices, linear algebra, curve fitting, integration, etc. The examples are in fact appropriate for scientists or engineers. One thing this book does not do is teach why these methods are used. You must know a bit about what these techniques are used for before reading this book.

Likewise, Miller does not teach Pascal programming here. Since there are so many good introductory Pascal texts available, that is not a serious problem. Since this book is not suitable for a first book, who would use it? It seems most useful either for self-study for the professional who wants to keep up to date, or as supplementary reading for a second programming course for technically oriented students. It might not be suitable as a primary text, because there are no formal exercises and solutions. However, there are many suggestions for further experimentation in the text. You should have access to a Pascal system to get the maximum value from the book.

The format of the book is very attractive and readable. The programs are presented complete in boxes, with the reserved words in bold face type. The

author understands that built-in functions like TRUNC or EXP are not reserved words and should not be printed bold face. That puts him ahead of some Pascal writers for accuracy! I found no typographical errors in the programs themselves.

Miller uses a good structured programming style, with consistent indenting for readability, and sufficient commenting. He does not use many advanced features of Pascal like defined scalar types, subranges, records, pointer types, packing, or recursion (except for a recursive version of Quicksort). Could this be because the programs are very similar to those in the author's *BASIC Programs for Scientists and Engineers*? As a result of the author's simple programming style, these programs will be useful for people who want to get started without spending a lot of time on details of Pascal, but it may not be suited for someone who wants to learn Pascal thoroughly.

The real strength of this book is the appeal to the intuition of the reader. There is a minimum of abstract math, and a maximum of applications. For example, most numerical analysis books tell the reader to beware of matrices with zero values on the main diagonal: simple Gaussian elimination won't work to invert them. Miller goes beyond merely recommending the method of pivots, he actually gives a complete, usable example. All of the programs are usable as is, and they are clear enough to serve as bases for more complex programs. The bibliography at the end includes most of the best works in programming and numerical analysis for those who want to go further. To sum up, this is a fine book for someone who wants to get a job done without a major investment in learning advanced techniques.

Intuitive IC Electronics
A Sophisticated Primer for Engineers and Technicians
by Thomas M. Frederiksen
McGraw-Hill
183 pages, $18.50
Circle No. 399 on Reader Service Card
Reviewed by George W. Jolly

A book that truly reveals mysteries to the uninitiated is a rare gem. *Intuitive IC Electronics* begins with the basic definition of an electric current, and takes you through the history of diodes, transistors, and integrated circuits. Mr. Frederiksen assumes no prior knowledge of semiconductors, and throughout the book he emphasizes pictures and plain English, not mathematics (thus the "intuitive" in the title). The mathematics in the book, if gathered together, would fill about six pages.

Intuitive begins with the physical nature of electricity. If terms like "thermal noise," "energy-band theory," and "avalanche breakdown" seem over your head, you probably should read this book. Mr. Frederiksen makes them seem simple, and sprinkles information about the physical construction of integrated circuits here and there to keep things interesting.

Intuitive uses the junction diode to explain many concepts used in other semiconductor devices. Bipolar transistors, JFETs, and MOSFETs also receive excellent treatment. The actual internal workings of these devices is fascinating. Mr. Frederiksen shows many small details to round out his presentation. Problems faced by the manufacturer of semiconductor devices are shown, some of which are not yet fully understood.

Mr. Frederiksen does not leave you with old information. Newer devices such as charge-coupled devices, large dynamic memory chips, and electrically erasable read-only memory receive excellent treatment. The book concludes with a short look at the future of the semiconductor industry.

Intuitive is a delightful book. Mr. Frederiksen has the rare ability to explain engineering concepts to non-engineers without boring us to sleep.

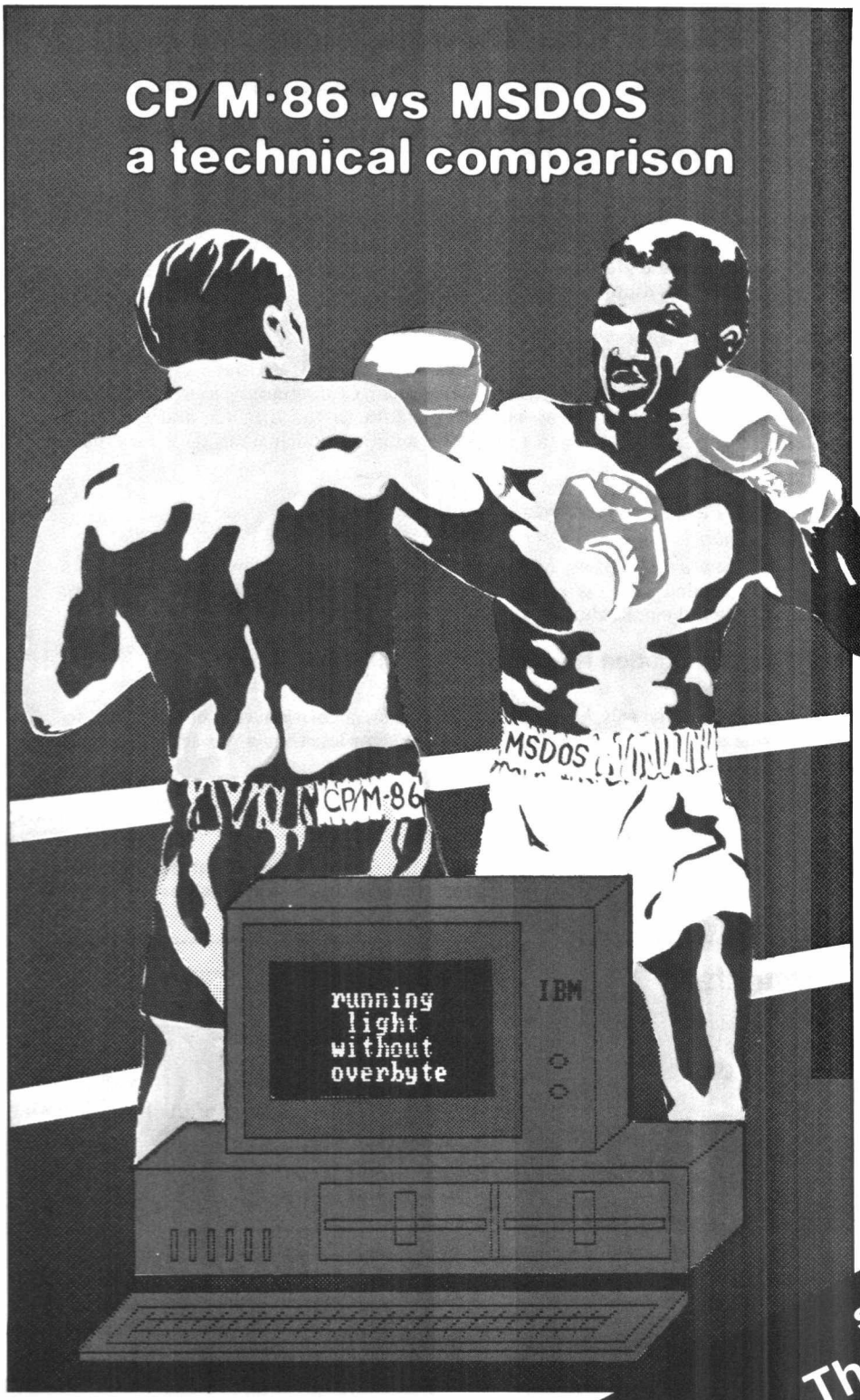
DDJ

$2.50
($3.00 in Canada)

DR. DOBB'S JOURNAL

For Users of Small Computer Systems

Number 69 July 1982



CP/M-86 vs MSDOS a technical comparison

Video Graphics on the IBM-PC

Approximation Algorithms

Software Reviews: Word Processors for IBM's Micro

and even more!

See our new feature!
The CP/M Exchange

DR. DOBB'S JOURNAL
For Users of Small Computer Systems

July 1982 Volume 7, Issue 7

CONTENTS

Editor – Marlin Ouverson
Assistant Editor – Fritzi Lareau
Contributing Editors –
Dave Caulkins, Dave Cortesi,
Ray Duncan, Gene Head,
Michael Wiesenberg
Marketing Director – Craig Harper
Advertising Manager – Laird Foshay
Circulation Manager – Gloria Romanoff
Circulation Assistants –
Theresa O'Rourke, Terri Marty
Art Director – Clifford West
Production Manager – Barbara Ruzgerian
Production Assistant – Margaret Tsai
Typesetter – Renny Wiggins
Cover Illustration – Nori Tolson

© 1982 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $40 for first class to Canada and Mexico; $58 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Thomas H. Grohne, G. V. Elkins, Robert M. Connors, William J. McEown, Robert C. Luckey, Robert N. Stabler, DeWitt S. Brown, John B. Palmer, Burks A. Smith, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Howard E. Decker, Jim Wilson, Frank Lawyer, Chris Pettus, Paul Lutus, Ben Goldfarb, Rodney Black, Steven Weiss, Pat Phelan, John Brodie, James Fleming, Unison World, Thomas Davis, Jan Steinman. **Sustaining Subscribers:** Pete Roberts, Steven Fisher Enterprizes. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Hofacker-Verlag, Tegernseer Strass 18, D-8150 Holzkirchen, West Germany. **Asia & Australia:** ASCII Publishing Co., Ltd, Aoyama Building 5F, 5-16-1 Minami Aoyama, Minato-ku, Tokyo 107, Japan.

Articles

14 CP/M-86 vs. MSDOS – A Technical Comparison
by Dave Cortesi
Author Cortesi sets two operating systems for the IBM Personal Computer head-to-head. They are measured against each other, as well as compared to CP/M-80. Amplifications, rebuttals, amendments and corrections sent by readers will be part of a follow-up article later this year.

32 Graphics on IBM's Personal Computer
by Ray Duncan
Not all details of the PC's high-resolution graphics are clearly documented in the official manuals. This article is confined to discussion of the color/graphics interface in hi-res text and graphics modes. Included are a memory map, line generator subroutine, graphics dump routine and more.

44 More on Checksums
by Robert Blum
DDJ #66 presented an article covering a number of commonly used check digit systems. This short piece serves as an addendum to the original, and may help readers perform a self-check for possible dubious conclusions they may have formed.

46 PDP-1802, Part II – Hardware Simulator
by Chris Hamlin
Last month saw a description of this simulator. The current installment presents all the code needed to allow a COSMAC 1802 to run PDP-8 code. Get it running – the author will be contributing some useful applications software as well.

54 Rough Approximation Numerical Algorithms
by H. T. Gordon
Underground lore reveals how an approximation algorithm can be designed to yield a value with either an absolute or relative error less than a pre-set value, over a defined *interval* of the numerical input. . . .

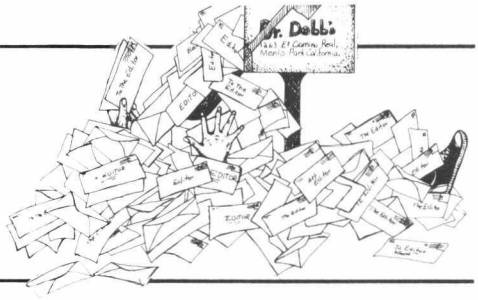
66 Small-C for the 9900
by Matthew Halfant
This powerful language puts down roots in the 9900, thanks to author Halfant. Follow him down the trail to his implementation on this 16-bit machine.

Departments

6 Editorial
"Dueling DOS'"
6 Letters
9 CP/M Exchange
10 Dr. Dobb's Clinic
57 Of Interest
61 Software Reviews
Word Processors for the IBM PC
76 Book Reviews

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, Box E, 1263 El Camino Real, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025 · 415/323-3111

LETTERS



IBM PC – No Nits is Good Nits

Dear Doctor:

Robert Baer's letter (He Picks IBM's Nits, March) really did surprise me with his nitpicking of the IBM Personal Computer. I just bought one myself for software development and am quite impressed with it. Some observations:

Few micros come with clock/calendars at all, especially as standard equipment. Mr. Baer doesn't like to tell his what day it is, but I don't mind. You have to tell a System/34 what day it is, too, so this must just be what IBM is used to. A particularly nice feature is the fact that the IBM PC dates each file in the directory with the date you saved it.

Yes, the FORMAT program is incredibly simple-minded. It must not verify since it seems to happily format just about anything. I agree it needs to be improved. There *is* a format function on the diagnostics disk provided with the system that can detect a bad disk quite well. It even reports track and sector where an error is found. The format this program writes is just for testing, but at least it will test a diskette.

I found some errors and omissions in the documentation too, but IBM's documentation is better than the competition. At least they can spell. They thoughtfully provide a couple of postage-paid "product comment" forms in the back of each manual so they must want users to tell them about any errors they find.

With a processor able to address 1000K of memory, one can be extravagant with it! Who cares if they want to reserve a mere 40K for system ROM? This computer uses more memory for system, DOS, and memory-mapped video than most "personal computers" have altogether with ten times more left over. I can't imagine what a single-user application would do with all that memory anyway.

The big disappointment is the totally inadequate 160K per drive disk storage, especially when you consider that reliable quad density 5" drives have been available for years. However, Tecmar has come out with a bus expansion unit and I know of at least one Winchester that is available now. I imagine the after market industry will soon jump in to fill the void with plenty of plug-in goodies.

Sincerely,
Burks A. Smith, President
Datasmith
Box 8036
Shawnee Mission, KS 66208

Editorial

Dueling DOS'

Not long ago, individuals who had thought of writing software for the new IBM microcomputer threw up their hands in disgust. The corporation had offered royalty terms insulting to prime developers, including a ceiling on the total amount of royalties that would be paid on any single piece of software, regardless of its sales. A combination, I suppose, of angry letters and deathly silence from the people they had counted on awakened IBM to their error. They had thrown open the door to third-party vendors with good compensation schemes.

Since then, IBM has sent notice that royalties will now be negotiated on an individual basis. This will give an edge to software sources experienced in negotiation and pricing, and allows the company to offer less attractive terms to novice authors with little or no track record. Such people may still find third-party agreements to be the most reasonable.

The moral, if any, is that even napping giants can be aroused by enough Lilliputian pinpricks.

* * *

Owners of IBM's Personal Computer have before them a short-lived dearth of software. It is a brief silence before the inevitable hurricane wall of products roars onto the scene. This is the moment of decision for many owners of (and most vendors for) that machine: which operating system will they support?

IBM's promotion of MSDOS will no doubt create an installed base of many thousands of users of that software. It fits well within Microsoft's ongoing strategy to become known as more than just the world's largest supplier of ROM Basic. As MSDOS' reputation evolves, it will surely establish public prejudice for or against their future offerings. Judging from the voracity of their marketing campaign, this fact has not escaped Microsoft management, either.

Meanwhile, back in Pacific Grove, Digital Research spent a few extra months developing CP/M-86, giving their competitor an unexpected lead time. Most predicted their finished product would be a logical extension of the pervasive CP/M-80, with enhancements or changes to take advantage of the 8088 processor but still compatible with existing software. Their development work had to take place at a time of internal reorganization, significant staff growth, and marketing strategies which struggled to accommodate their newest corporate acquisitions.

Finally, both systems are available; it remains to be seen how software vendors will react to them. Because neither product will ultimately be confined to one machine, the comparison contained in this month's *DDJ* carries considerably more weight than would otherwise be true. We have given a good deal of space to a topic which will become of even greater relevance with the passage of time.

We have tried to keep our comparison on an objective, technical level by simply lining up similarities and differences side by side. The reader may draw his own conclusions. By all current indicators, if a clearcut winner emerges from the confrontation of the two operating systems, it will be a winner by decision and not by knockout.

– Marlin Ouverson

Dear Dr. Dobb,

I have several comments on the March 1982 issue of *DDJ*. Perhaps the most important of these comments concerns the Editorial (page 3). The Foundation trilogy was written by Isaac Asimov, not Heinlein.

Regarding the letter from Robert Baer on page 4, I would agree with Nit #1 (DOS demands a date upon power-up) only if the date is not used for anything. If this date (and time?) is, for example, put on listings by a list utility and/or if the date/time of the last modification to a file is stored on diskette along with the file, then I see little reason to object.

On Dr. Dobb's Clinic (page 6), I believe that Microsoft BASIC for the TRS-80 (Level 2) uses Method 1, not Method 0, for searching for the target of a GOTO command. (Method 0 is always start at the beginning of the BASIC program; Method 1 modifies this to start at the current position if the target line number is higher than the current line number.)

Finally, regarding 8080 Pidgin (page 24), since Mr. Bender asks "How about an 8086 version? I will ask "How about a 6809 version?"

Sincerely,
Jim Howell
5472 Playa Del Rey
San Jose, CA 95123

A BIOS for Epson

Dear Marlin,

Gary Novosielski's December article "For CP/M BIOS An Improved MX-80 Printer Driver" was very commendable with respect to the depth of his observations and the clarity of his reporting. In particular I think he did an outstanding job of documenting his driver program.

This letter is printed on my MX-80F/T. When I read the article I had recently acquired the printer, and since a major reason for its purchase was my need for a high performance unit, I found his findings to be somewhat on the discouraging side. While I can follow his program with some effort, I am not yet in a position, assembly-language-wise to tackle a major modification on my H89's BIOS.

I am using The Software Toolworks PIE editor and TEXT formatter. I had observed in the manual that instructions for underlining and emboldening were provided, also references to backspacing. Having Gary's article in mind, and a feeling that word processors were really meant for Diablos or the like anyway, I didn't give the possibility a whole lot of attention. Then a few days ago a situation came up for which I required the use of both these features. So what the heck, I figured, I'll give it a try and see if Gary is really right.

He is, in spite of the appearance of this letter. I am delighted, because my less than seventy dollar word processing software is able to cope. TEXT lays down the underline and/or the text to be emboldened at the required position(s) in the line on a first pass without a line feed, then returns to lay down the line in its entirety. Backspacing is available as a secondary procedure when for some reason the primary is unsuitable.

The approach taken by PIE and TEXT is not, I think, the real thing in utilizing the Epson's control features. But an enhancement is provided that I could have been enjoying much sooner had I but looked more closely at what I had. Which is the point of my writing. Don't make assumptions.

Regards,
Clem Pepper
3270-96 Caminito East Bluff
La Jolla, CA 92037

All in Favor . . .

Dear Doc,

Thank you for the interesting articles on TINCMP and PIDGIN. Allow me to second Mr. Herrold's request (Letters, *DDJ* #67) that you publish the programs to test the PIDGIN macro definitions.

I am currently implementing TINCMP and PIDGIN for a cassette-based H8. If, by any chance, any of your readers are interested, I would be glad to pass the results of my work along to them.

Thanking your for your consideration, I am,

Yours truly,
Terrence Enger
150 Graydon Hall Drive, #1207
Don Mills, Ontario M3A 3B2

P-Code vs. CP/M

Dear *DDJ* and readers:

Following SofTech's recent announcement of an 8086 version of the p-System, I contacted them for information on upgrade conversion from my present 8080 copy of version IV. I was informed that the p-System for the 8086 was a separate product, and that a complete system would have to be purchased under a new license.

When I originally purchased the UCSD system, I was under the impression that by committing to an interpreted system I was trading execution speed for portability, thus protecting myself against obsolescence as newer processors became available. The literature of the time (1979) indicated that only an interpreter change would be needed when changing CPUs. Since then, new p-machines have replaced old more rapidly than new processors have appeared, and each new version entails another charge to keep licenses current.

Now I find that although my hardware is processor-independent, and in fact currently employs three coprocessors, and although SofTech now supports native code generation for them (although crudely), it is practically infeasible to port my existing system software from the 8080 which presently executes the p-System to the 8086 which performs only background I/O. To *use* my 8086, I am looking at an even $1000 to SofTech alone in converting my operating system, compiler, and utilities. A kilobuck, to gain minor execution-speed improvements on a minority of my programs, is rather steep. I'd get more by keeping the 8080 interpreter, but with added support for my 9511 math chip. Unfortunately, SofTech doesn't want us doing *that*, either.

At present, no other vendors support version IV with anything useful, so I use version II for everything except my own reinventions of the wheel. The widely proclaimed source compatability is, of course, useless to those of us without source licenses, and merely gives us the option of scrapping our software investments every year or two in order to use functionally *identical* programs.

By comparison, I have a CP/M-86 program which interprets CP/M-80 programs quite nicely, even though they are object-only native code. Normally I have no need to interpret, since I can run CP/M-80 in the 8080 processor, just as I do the p-System, but it is convenient not having to shuffle systems. My CP/M programs written for version I.3 prior to 1978 are therefore alive and running on an 8086 in 1982, but my p-code programs

(Continued on page 72)

CP/M EXCHANGE

by Gene Head

Many *DDJ* readers go back at least as far as ALTAIR and IMSAI front panels, with recollections of toggling binary bits to enter a boot loader and then loading the operating system from a Tarbell cassette interface. Without a simple keyboard monitor in ROM, debugging in those early days took a very long time. One slipped bit usually meant reloading by hand the boot loader followed by reloading the operating system cassette followed by the editor or BASIC program, etc., etc.

Before long, PROM and EPROM keyboard monitors were available costing about $200. After a crash or on power-up, keyboard commands could be used to load a cassette program, examine and fill memory and a few other less exotic functions. We geared ourselves to the Tarbell cassette and there were actually published programs for business applications using cassette. Hard to imagine running a payroll program using three or four passes on a cassette! Floppy disks were being used by a few of the hardware experts (sometimes amateurs but still experts) but most of us only dreamed about the affordable floppy.

Time passed. It seemed like forever. The cost of the floppy remained out of reach of most hackers. Then the breakthrough – a cut-down version of the floppy disk called a mini-floppy appeared in magazine ads. The corner was turned. High-speed loading, with storage in the kilobytes was a reality. Technology has continued improving until today we have several megabyte hard disks and virtual disks (lots of RAM that is accessed like a disk one "track" and "sector" at a time) that cost just a little more than the original mini-floppies.

Digital Research gave us CP/M with the main advantage of device I/O independence. It doesn't matter to the CP/M programmer if the printer is a 110 baud TTY or a 9600 baud parallel printer. Programs became transportable! But what about the actual transportation?

The technology that brought us out of the dark ages of cassette interfaces also created a problem that has never been overcome. No standard for mini-disk recording was adopted. Exchanging disks to pass data from one computer to another is quite impractical. With soft- and hard-sectored disks, single- and double-density disks, single- and double-sided formats available, exchanging data between non-identical systems is impossible. One vendor of software lists no fewer than twenty different formats for mini-disk distribution.

Digital Research and Lifeboat Associates have done their best to keep the CP/M defacto standard just that, *the standard*. CP/M formatted data can still be exchanged from my North Star computer to my friend's Superbrain computer *if* we can find the method of transporting the data. Once I have the CP/M-formatted file of data I can access it through a similar CP/M program. But how do I get it from the Superbrain to the North Star?

Computer-to-computer communication is one answer.

The modem, and future telecommunication devices, will bring us out of the non-compatible "boxed information" dilemma and into the realm of instant information access and rapid information exchange. Floppies gave us mass storage with rapid access. Telecommunications will broaden the field of access and give us rapid *exchange* of data.

The Pioneers

Like a lot of hackers, I started out in Ham radio. After thousands of radio contacts I couldn't see much thrill in leaving a message on a Computer Bulletin Board. However, when I found a disassembler that provided a much-needed utility to my software library, and that I could transfer for the cost of a phone call, I knew there was more to modem communications than creating digital graffiti.

Ward Christensen has given us the CP/M modem program, MODEM. This program allows for the transfer of CP/M files between users connected via phone line and modem. XMODEM, as it is named on most every remote CP/M system, is clearly the standard for modem information exchange. A simple system of sending data followed by a check-sum assures accurate and efficient data exchange. XMODEM is very friendly and even a novice will be able to transfer files with very little head scratching. If you don't own a modem, figure out a way to get one. Buy one cooperatively with friends or build one or borrow one. There are literally hundreds of remote CP/M systems with programs and data available for the cost of a phone call!

Kelly Smith in Simi Valley, California, has the largest library of public domain software available to modem users I have seen. On my last connect I counted 1628 separate CP/M files, in just under 10 megabytes of disk space. These files included BASIC business programs from Adam Osborne, MACRO Libraries for programmers, utilities to test memory and disks and CPUs, and source code for languages like FORTH and STOIC, not to mention games and graphics.

Information exchange is happening. You're missing out if you don't get in on it!

About This Column

Dr. Dobb's will dedicate space regularly to the CP/M user, the novice as well as the veteran. There should be enough room to address many levels of expertise. We have a unique opportunity to really communicate effectively. Thanks to the narrow scope of interest here, we will all be speaking the same terms and jargon. I know some of the readers of this column can get their hands on a modem, if only to borrow one for the weekend. This will allow practical exchange of solutions to common problems. Finally, I *think* I will be able to handle individual communications. Your questions and comments will be serviced in one of two ways.

1. Your SASE will get you the name of another person who can solve your problem or share information.
2. Your comments and/or questions published here should bring a direct response from fellow readers.

I realize I could be biting off more than I can chew, so if this gets out of hand I reserve the right to cancel the one-to-one stuff and go on to editing your comments and contributions.

You will determine the content of this column, not me. I'll collect your ideas, edit your contributions and maybe throw in some thoughts of my own. However, this is not my column, it is *our* column. If there is not enough interest on your part to maintain it, we will scuttle the whole thing very quickly.

What Exactly Is "Interest"?

If you were writing this column and I was reading it, I would make the following contribution in the form of a letter:

> Dear Gene:
>
> Every time I boot my CP/M system it hangs up if the printer is not "ON" and "ON-LINE". How can I fix this? Sometimes I have

(Continued on page 67)

DR. DOBB'S CLINIC

by D.E. Cortesi

Old Business

We continue to get a trickle of letters on calendar matters. Who is Gregory, when and why did he change the calendar, and by how much; who is Julius and what kind of dates does he get? Readers who are interested in these things can now refer to two good survey articles in other journals.

The first appeared in that excellent British publication, *New Scientist*. In the issue dated March 25, 1982 (Volume 93, Number 1298, page 770), Geoffrey Thomas nicely summarizes the astronomical basis of the calendar, the problems Pope Gregory XIII addressed, and the present state of things. If you aren't familiar with *New Scientist*, you have a pleasant discovery in store. Among other things, it carries frequent (if superficial) coverage of European developments in personal computing. Ask your library to stock it.

The May, 1982 issue of *Scientific American* contains an article by Gordon Moyer that covers much the same ground, but with more detail and in greater historical depth (and with the heavier writing style typical of *SciAm*).

A BASIC Misunderstanding

David Levy of Blacksburg, VA, writes to tell us of what he regards as an error in IBM Personal Computer BASIC. "There is a confusion in the boolean operations when lines are broken by colons," he says, and demonstrates it with this complete program:

```
FOR I=0 TO 99:IF I MOD 2 THEN
  ?I:NEXT
```

which prints exactly nothing, while this apparently identical program –

```
10 FOR I=0 TO 99
20     IF I MOD 2 THEN ?I
30 NEXT
```

– prints 50 odd digits as you would expect. The same "bug" can be displayed using WHILE-WEND instead of FOR-NEXT. Has Levy found a glaring error in PC BASIC? Think about it.

We think not. Two obscure, poorly documented, effects are operating in Levy's program. Their interaction makes the program's behavior seem very strange, but it isn't wrong.

The first thing is that, in all Microsoft BASICs, the scope of an IF is delimited only by the keywords THEN and ELSE and by the end of the line. Note, *line*, not *statement*. Colon delimiters and other keywords *do not* delimit the parts of an IF. This is spelled out in the IBM manual (page 4-100), implied in the TRS-80 Level II manual, and not said at all in the BASIC-80 5.0 manual.

The second thing is that an END statement (or the end of the program text, which has the same effect) discards all open loops without an error message. One would expect that omitting line 30 of the second example would result in a "FOR without NEXT" error, but it doesn't. This is not stated in any of the three manuals we checked.

What happens in Levy's program is that a FOR loop is begun; the value of I MOD 2 is tested; it is zero and so the true-scope of the IF is skipped entirely. That takes the interpreter to the end of the line (*not* to the colon!). It proceeds to the next line, but there isn't one. So it tosses the open loop into the bit bucket and, with an innocent smile, says "Ok."

It's just one of the little things that makes it such a delight to teach beginners to program in BASIC. Or, as Dr. Edsger Dijkstra says in the May issue of *SIGPLAN Notices*, "It is practically impossible to teach good programming to students that have had a prior exposure to BASIC; as potential programmers they are mentally mutilated beyond hope of regeneration."

Root And Branch

In the March issue we published a long discussion of the possible ways that a BASIC interpreter could execute a branch. A number of readers responded.

Some commented on programming style. In general, two attitudes emerged: one was that there are always programs that need, and programmers that will use, whatever quirks a language allows – and so what? The other was that, BASIC being such a poor language for building large programs, it's pointless to worry about subroutine placement.

Well, yes, but our comments were a reaction to quite a number of naive write-ups in computer magazines which would be read by even more naive programmers. Here's a recent example, typical of many: "Thus the second lesson is to place subroutines as close as possible to the beginning of the program. Some of your programs that heretofor seemed to drag on unmercifully could now have more zip" (from "Considering Space And Time In The Atari," C. M. Levy and G. Levy, *Compute!*, May, 1982). Our point was that such advice is terribly ephemeral. Levy and Levy are correct, for the Atari of 1982. But programs, and ideas, live much longer than the systems that spawn them. How many readers of *Compute!* will still carry that "lesson" three years from now when they are using their 256KB, 32-bit-CPU, Atari 9000s?

Two readers noted that the TRS-80, at least, searches forward from the present line when the target line number is greater. This is almost our Method 1, but it appears that the TRS-80 starts from the top of the program when the target line is less than *or equal to* the present line number. This is surprising; it seems obvious that a branch to the present line, as in

```
1000 IF INKEY$="" THEN 1000
```

should execute in minimum time, but Microsoft didn't see it that way.

Several readers pointed out a method that we didn't consider. Tom Lane of Pittsburg, PA, describes it best.

"When a line is tokenized, replace line numbers with the sequence:

(1) The line-number-follows token (a byte),
(2) Two zero bytes (a pointer to the target line will be placed here, later),
(3) The line number itself, converted to binary.

At RUN time, scan the text looking for line number tokens. For each one found, pick up the line number value, find the referenced line (using Method 0 or 1), and install a pointer to that line in the reserved space. If the line is nonexistent, put in a zero.

"To execute a branch, just pick up the pointer from the token, with a check for zero. Aside from eliminating the computation and search costs, this method also eliminates the need to convert the text form of the line number to binary during each execution of a branch. It should be at least a factor of 10 quicker than your methods."

We assumed that line numbers would be stored as binary tokens; sorry if that wasn't clear. Lane also corrected our analysis of Method 2. The expected number of iterations should have been $3n/16$, not $n/6$.

This is certainly a workable method. A line-number token must contain both

a number and an address, as Lane suggests, not just an address as suggested by several other readers. The reason is that the interpreter must have some way of telling when the target line does not exist. An interpreter has to be able to tolerate the presence of such branches, diagnosing them when they are executed.

That method allows only static line references. Dynamic references (naming line numbers as variables rather than constants) can be quite useful. When the line search is driven by line numbers, the interpreter can handle a branch to a variable. When the search relies on addresses, the interpreter has to require every branch target to be given as a constant. The Microsoft interpreters require static references, but take no advantage of the predictability that results.

A Designer's Experience

The best response came from Bill Wilkinson of Cupertino, CA, a fellow columnist and the designer of several BASIC interpreters. All that follows is extracted from his letter.

I don't believe anyone still uses Method 0, the straight linear search through memory. Certainly even simple-minded Atari BASIC uses a scheme similar to Method 3. [An Atari line is:

NNNN LL SS tokens SS tokens . . .

where NNNN is the line number in binary, LL the line length, SS a statement length.] The line length is a single byte which means that lines can't be longer than 255 bytes. Big deal.

Note the use of statement-length bytes. Most BASICs find these unnecessary, but when we designed Atari BASIC we wanted to make it as flexible as possible. So we allowed the user to STOP, change his program (!), and GOTO or CONTinue without losing track of FOR . . . NEXT loops, GOSUBs, or variables. The consequence: the run-time stack can't contain absolute addresses, since the process of changing the program may very well change a line's address. Instead the stack contains a line number and a statement offset within the line. Hence the statement length bytes allow us to scan to the appropriate statement quickly.

Cromemco 32K Structured BASIC

The user types in a source line. BASIC scans it for correct syntax, converts it to internal token form, and checks to see if it was preceded by a line number. For now, assume that this is a new line number. The tokenized line, *with* statement length bytes but *without* its line number, is simply added to the end of the program space. Its line number is added to a line number table where it is linked [in line number sequence] into a singly-linked list of all line numbers entered so far.

You might presume that having a line number table is equivalent to Method 8, a hash table built when the word RUN is given. Not so. With Structured BASIC, the tokenized version of GOTO 100 looks something like this:

GO L# TTTT

[where GO is the GOTO-token, L# is the line-number token, and TTTT is an address in the line number table]. The address is actually the relative address within the line number table for line 100. The line number table need not be searched at all, except when the line is first entered. Within the line number table [each entry appears thus:

AAAA LLLL FFFF

where AAAA is the address of a line in the program space, LLLL is its line number in binary, and FFFF is the relative address of the next line number table entry in sequence]. The execution scenario of 100 GOSUB 200 follows:

This is a GOSUB token. Push the current line number table address (*not* the line number, *not* the line address) and statement offset on the run-time stack. Now pretend we are a GOTO instead.

Is the next token a line-number-follows or a line-label-follows? It has to be one or the other; the syntax checker (which is also the tokenizer) won't let anything else through. Aha; it's a line-number-follows.

Use the next two bytes as a relative address pointing to an entry within the line number table. Get the two bytes [AAAA, above] thus pointed to. These two bytes are the address of line 200 in the program space. Start execution there. That's it! No scans, no hashing, just two indirect address loads.

[There are several consequences of using a line number table.] Line labels: Structured BASIC allows code like this:

```
100 Goto Totalpay
. . .
1370 *Totalpay : . . .
```

Labels are treated much like variables and are placed in the variable table (which is also built at entry time, not at run time). A special flag in the table says "this is a label," and the value of a label is the address of its corresponding line within the line number table. So a label costs only one extra indirect address load. (Variables are tokenized as addresses within the variable table, and can be used with no space or speed penalty.)

Renumbering: To renumber a program we have to change all references to each line number as well as the line number in the line itself. Wait a minute; what references? What line number in the line? We simply resequence the line number table and all is done!

Deleting lines: At last, some overhead. When a line is actually deleted, its space must be recovered from the program table. So all succeeding lines must be moved "down" in memory. Then we must go traipsing through the line number table changing the addresses of lines which were higher in memory than the deleted line. But we do not have to go in linked-list order, so the process is reasonably fast. Replacing an existing line is done by first deleting it and then adding it to the end of the program space as if it were a new line.

Unfortunately, deleted lines remain in the table (they must, or we would have to do drastic patching on the table). But there's a workable solution to this problem: LIST the program to disk and reENTER it (from disk), a not too time-consuming process. [*Bill, why couldn't you link the freed table entries into a chain and reuse them as new lines are entered? DEC*]

Relocation: Each program consists of five parts: the Program Space (tokenized lines), Variable Table, Line Number Table, String/Array Space, and Run-Time Stack. Any reference to an element in one of these spaces is nearly always by address. But since the various spaces might grow, shrink, or relocate, all addresses are relative to the start of the appropriate space.

For more details, see my paper in the proceedings of the Second West Coast Computer Faire. That paper discusses Cromemco 16K BASIC, but the principles are identical.

DDJ

CP/M-86 vs. MSDOS

A Technical Comparison

Editor's Note: We think that this article may stir some controversy among our readers. If a significant number of readers are moved to respond to it, we will publish a follow-up article in a later edition. The follow-up will air the corrections, amendments, amplifications, and rebuttals of respondents.

Users of personal computers based on the Intel 8086 (or 8088) have a wide choice of operating systems, but remarkably little technical information on which to base a choice among them. In this article I intend to present a detailed technical comparison between the two best-known operating systems for these machines: CP/M-86 from Digital Research, and MSDOS (or IBM DOS, or SB-86) from Microsoft. There are other operating systems for the 8086 – the UCSD P-system from Softech, and Oasis from Phase One Systems, to name two – but I am not competent to review them.

Sources

I got information on the systems from several sources. My primary sources were vendor manuals. The IBM *Disk Operating System* and *Technical Information* manuals were my primary sources of information about MSDOS. These manuals are well-written and handsomely presented, but I found them to be somewhat stingy with detailed technical information on MSDOS.

I supplemented my reading with two weeks of full-time use of an IBM Personal Computer equipped with MSDOS version 1.0, on which I wrote several programs both in assembly language and in BASIC. I also had access to the proceedings of a Microsoft seminar on MSDOS. This material mentioned several MSDOS utilities which were not in fact present on the IBM distribution disk. This article is based on MSDOS as distributed for the IBM machine.

The *CP/M-86 User's Guide*, *CP/M-86 Programmer's Guide*, and *CP/M-86 System Guide* from Digital Research were my sources for that system; I also had a brief look at the IBM version of the CP/M-86 manual. These manuals bear no resemblance to the old CP/M-80 books. They are clearly the work of professional writers and are very good indeed. If Digital Research would spring for professional typesetting and layout, their new books would bear comparison to any.

Information on the 8086 came from Intel's *8086 Family User Manual* and from Steven Morse's excellent *8086 Primer* (Hayden Book Company, 1980).

CompuView Products, Inc. generously provided an evaluation copy of their version of CP/M-86 for the Personal Computer. I spent several hours with that system as well. Unfortunately it was a preliminary version which had not been completely shaken down. Except for a few special features, I didn't factor the CompuView experience into this article.

Objectivity

Many people equate the term "technical" with the word "objective," and that with "fair" and "unbiased." It would be nice if a technical comparison could automatically be an objective one, but there is really no such thing as a value-free judgment. I have attempted to present facts in an objective way, and all apparent statements of fact that follow are based either on vendor literature or on actual machine experience.

Whenever I draw an inference from the facts, or compare two sets of facts, you may assume that my conclusions are affected by my tastes and interests and so are not value-free. I have marked such extra-factual assertions with qualifiers like "I think," or "it seems likely," so you can tell the points at which you may want to reach a different conclusion.

by Dave Cortesi

Copyright © People's Computer Company 1982. All rights reserved. Permission is hereby granted to reproduce this article for any non-commercial purpose. Reproduction, in whole or in part, for commercial advantage is prohibited.

Overview

It seems clear that both systems were modeled on CP/M-80. They offer the same command language, support the same file services, and have the same general structure, as that well-known system.

CP/M-86

CP/M-86 is a product (and trademark) of Digital Research, Incorporated, the company that produced CP/M-80. Like its parent system, CP/M-86 is composed of four parts: a Basic I/O System (BIOS), a Basic Disk Operating System (BDOS), a Console Command Processor (CCP), and a set of commands.

The BIOS masks hardware dependencies from the rest of the system. It must be provided by the system integrator – the vendor or hobbyist – and is usually written in 8086 assembly language. The BIOS provides a low-level, hardware-independent, interface to the disk system, and byte-level access to four logical serial devices. The mapping of the logical devices onto physical ports may be fixed, or the BIOS's author may choose to support redirection of serial I/O through the IOBYTE mechansim.

The BDOS provides I/O services to applications programs and to the CCP. It provides file access services, and it manages file directories and disk space allocation. It also offers services for the allocation of transient storage, and contains the logic needed to load a program from a disk file.

The CCP creates the user interface to the system. It reads command lines from the logical terminal or from a command file, parses them, and carries them out. It contains the logic of a few commands and executes others by loading programs from disk.

MSDOS

MSDOS was commissioned by IBM and produced by the Microsoft Corporation from a base written by Seattle Computer Products, Inc. Lifeboat Associates, an important software vendor, has undertaken to market the system for other 8086-based machines, and to encourage the development of application programs for it.

MSDOS is composed of a BIOS, a BDOS, a command processor, and a set of commands. The BIOS has much the same purpose as that of CP/M-86. In the IBM Personal Computer, the BIOS is in ROM. On other hardware, it will presumably have to be supplied by the system integrator.

The BDOS provides I/O services to application programs and to the command processor. It manages file directories and disk space allocation. It does not contain the logic to load pro-

grams from disk, nor does it provide services to allocate transient storage.

The COMMAND program creates the user interface to the system. It reads command lines from the terminal or from a command file, parses them, and carries them out. It contains the logic of a few commands and executes others by loading them from disk.

The User Interface

A user interface is composed of many things: the syntax rules of the commands; the methods of entering and editing command lines; the conventions used to name files and devices; the commands supplied with the systems; and the method of executing commands in batch mode.

Table 1

Line-Editing Control Characters

Edit Function	CP/M-86	MSDOS
Destructive backspace	backspace	backspace
Tab to next 8th column	tab	tab
Mark end of line, type clean line, continue	ctrl-R	F5 then F3
Mark end of line, discard input, start over on next screen line	ctrl-U	ESC
Continue input on next screen line	ctrl-E	LF
Terminate line, deliver to program	CR or LF	CR
Clear current screen line, discard input, start over on current screen line	ctrl-X	none

Table 2

MSDOS "Template" Editing Keys

Edit Function	Key
Copy down next character of prior input line	F1 or →
Copy characters of prior line up to "x"	F2 then x
Copy all remaining characters of prior line	F3
Skip one character of prior line	DEL
Skip characters of prior line up to "x"	F4 then x
Make current input the "prior" line, mark end of line, start over on next screen line	F5
Toggle "insert mode," in which invisible pointer to prior line does not advance with the visible cursor	INS

Command Syntax

Both systems use the syntax of the CP/M-80 command language. Tokens are sequences of nonblanks delimited by blanks and other characters. The first token on the line is the command verb, either the name of a routine internal to the command processor or else the name of a file. The systems have identical rules for the formation of file references. Both manage the file system on a disk-drive basis; both use single letters to name the drives; and both assume a default drive which qualifies all file references where the user fails to override it. Both prompt for a command with "*d*>" where *d* is the drive-letter of the default drive.

MSDOS improves on CP/M-80 in one minor respect. It accepts the tab and the comma, as well as the blank, as a token delimiter. It also accepts the semicolon as a general delimiter; that introduces an incompatibility with MP/M II, in which the semicolon delimits a file password.

Command Entry and Editing

The systems have similar rules for entry and editing of command lines. The keys used for editing are summarized in Table 1. MSDOS has more elaborate support for console line input than does CP/M-86. It retains the last input line as a "template." Function keys let the user draw down parts of this template for use in the current line of input (see Table 2). It remains to be seen how these keys will be implemented on a non-IBM keyboard.

In my opinion the "template" is an excellent idea, but I found it difficult to use in practice. The problem is that you have to manage two different cursors: the one you can see in the current line, which moves in response to ordinary keystrokes, and an invisible cursor that moves over the prior input line under the control of the function keys. The function keys that move the invisible cursor have different uses when BASIC is in control.

The CompuView version of CP/M-86 for the IBM P.C. goes well beyond this feature. It lets you edit any line visible on the screen, much as you can do when working with IBM (or Atari, or PET) BASIC. Not only the prior command but any line – even a line of program output – may be edited and re-entered. I found this feature a delight to use.

Conventional Names

Table 3a shows the conventional names used for serial devices in CP/M-86. These names are used only as operands of the STAT and PIP commands. The MSDOS device names (Table 3b) may be used anywhere a filespec is permitted. An MSDOS program may open the "file" named CON: and so do console I/O using the BDOS services for file access. I will return to this important point.

Table 4 shows the conventional filetypes used in the two systems, in so far as I could determine them. Table 4 may be incomplete, and of course each software package brings another convention with it.

Command Invocation

In both systems, the first token of a command line becomes the argument of a search for the code that implements that command. Both systems improve on CP/M-80 in the way this search is conducted, but in different ways.

In CP/M-86, the CCP searches its list of built-in commands first. If that search fails, it looks for an executable file on the default drive under the current user (i.e. sublibrary) number. CP/M-80 gave up when that search failed. CP/M-86 tries once more, this time searching the default drive under user number zero for a file with the SYS attribute. This last search makes the user number feature, introduced with CP/M-80 2.2, useful at last. You can store common commands under user number zero, yet still invoke them when working under another user number.

The MSDOS search procedure is similar for the first two steps. COMMAND checks first for a built-in command. Then it searches the default drive for a file of type .COM. When that fails, it searches the default drive again, but for a file of type .EXE (relocatable object code). If the command still hasn't been found, COMMAND searches once more, this time for a file of type .BAT – a text file of command lines. If that is found, the commands in the file are executed in batch mode. I think this is a great convenience. You can write a .BAT file and then use it like any other command. It's not a new idea (mainframe systems have used it for years) but it's a welcome one.

Incidentally, I had to verify the MSDOS search hierarchy by experiment. The IBM documentation does not state which filetype has priority. The point is significant because, since .BAT files are last in search order, it is impossible to enhance a command by encapsulating it in a batch command file of the same name. Nor is there any point in creating a .EXE program with the same name as a .COM program; the latter will always be found first.

Table 3a

Logical Device Names in CP/M-86

Name	Designated Device
CON:	assigned console device
LST:	assigned printer device
AXI:	assigned auxiliary serial input
AXO:	assigned auxiliary serial output

Table 3b

Logical Device Names in MSDOS

Name	Designated Device
CON:	terminal device
PRN:, LPT1:[1]	printer device
AUX:, COM1:[1]	first serial port

[1] The corresponding xxx2: device is not recognized as a filespec.

Table 4

Standard Filetypes

File Contents	CP/M-86	MSDOS
Backup file created by an editor	.BAK	.BAK
BASIC program	.BAS	.BAS
Batch job (file of commands)	.SUB	.BAT
Loadable machine-code program	.CMD	.EXE[1]
Printable file (assembler output)	.LST	.LST
Object program (assembler output)	.H86[2]	.OBJ[3]
Assembler symbol table	.SYM	.CRF
Temporary file	.$$$	.TMP[4]

[1] .COM files also exist, but there seems to be no way to create them (perhaps with DEBUG?).

[2] Non-relocatable.

[3] Relocatable.

[4] Inferred from a single example (LINK).

The Command Sets

Each system comes with a set of commands. Some are built into the command program; the rest are executable programs loaded from disk. When comparing the two command sets, keep in mind that in both systems the command sets are open-ended. A missing command is simply an opportunity for an independent software vendor.

The CP/M-86 commands are summarized in Table 5. In the Type column, "B" stands for "built-in" and "T" means "transient." Commands marked with an asterisk are documented only in the IBM manual for the system. They support unique P.C. features (CompuView supplies some similar functions, but as part of a general configurator command).

There are several changes from CP/M-80. The DIRS, COPYDISK, and HELP commands are especially welcome. The PIP command will now copy to, as well as from, a user number other than the active one – another step toward making the user-number concept useful.

I have doubts about the TOD command. It is described in the *User's Guide* with no mention of hardware requirements. It definitely exists in the IBM version of the system. But the *System Manual* says nothing about its support requirements; there is no BIOS entry for clock operations; and there are no BDOS functions to set and read the time. I infer that the TOD command and the system clock/calendar are not device-independent functions of the system. Rather, TOD must be an optional nicety supplied by the system integrator to the pattern of the documentation.

The MSDOS commands are summarized in Table 6. There are a number of differences from CP/M-80; some are improvements. The DIR command, in my opinion, is not one. It displays the creation date and size of each file, but it shows files

only one per line. It took me only a few days to collect more than 24 files on one diskette; from then on I had to hold the screen when I used DIR. The CP/M-80 approach of separate summary (DIR) and detail (STAT) displays seems better to me.

The RENAME command does improve on CP/M's REN. Even after two years, I still get the operands of REN backwards. RENAME takes operands of *oldname*, *newname*, and where a letter of *newname* is ambiguous, the corresponding letter of *oldname* is copied. Thus the command "rename alpha.bas *.asc" renames "alpha.bas" to "alpha.asc" with minimal keying.

Note that COPY, the general file-transfer utility, is a built-in command. This is an improvement; it makes that frequently-used command faster, and removes the need to have a copy on every disk. Unfortunately, COPY is not as capable as PIP. It lacks the functions of PIP's options D, E, F, L, N, P, Q, S, T, U, V, and Z (the functions of PIP options G, H, I, O, R, and W are, for one reason or another, irrelevant in MSDOS). Nor can COPY concatenate files. Its lack of a verified-copy option is partly compensated by the presence of the COMP utility (but note that COMP is a transient). Similarly, DISKCOPY does not verify, but the DISKCOMP utility can be used to verify its work afterward.

After living with them, it is my opinion that some MSDOS commands are unduly chatty. The COMP, DATE, DISKCOMP, DISKCOPY, and TIME commands all require a user response even when all their operands are given on the command line. Some of these responses are needed to support single-disk systems (but are required in a two-disk system regardless); others just seem officious. Because they demand keyboard input, those commands cannot be used in an unattended batch file. This limits the utility of MSDOS's fine batch facility.

Batch Command Files

CP/M-86 provides exactly the same facility for batch execution of commands as do CP/M-80 and MP/M. The SUBMIT command converts an editable file of command lines into an internal form. Before prompting the user for a command, the CCP looks to see if the file $$$.SUB exists on the A: drive; if so, the CCP takes the next command from it rather than from

Table 5

CP/M-86 Command Functions

Command	Type	Function	CP/M-80 equivalent	MSDOS equivalent (comment)
DIR	B	display summary of directory	DIR	DIR (detailed info, one file/line)
DIRS	B	display summary of system files	none	none (can't display hidden files)
ERA	B	erase directory entries	ERA	ERASE
REN	B	rename a file	REN	RENAME (easier syntax)
TYPE	B	display file at console	TYPE	TYPE
USER	B	set/display sublibrary number	USER	none (no sublibrary concept)
ASM86	T	assemble 8086 code	ASM (8080)	none (assembler is separate product)
*ASSIGN	T	assign logical:physical devices	STAT	none (logical:physical mapping fixed)
COPYDISK	T	verified track-level diskcopy	none	DISKCOPY (does not verify)
DDT86	T	debug at machine-code level	DDT	DEBUG (no assemble command, but disk I/O)
ED	T	line-oriented editor	ED	EDLIN (many fewer commands)
*FUNCTION	T	assign strings to Fn keys	none	none (DOS uses keys for template edit)
GENCMD	T	make .H86 file executable	LOAD	LINK (a full relocating linker)
HELP	T	on-line command hints	none	none
LDCOPY	T	copy bootstrap tracks	SYSGEN	SYS, FORMAT
*NEWDISK	T	format a diskette	(vendor)	FORMAT
PIP	T	general file transfer	PIP	COPY (many fewer options)
*PROTOCOL	T	set asynch-port protocol	none	none (with asynch support code?)
*SPEED	T	set asynch-port bit rate	none	none (with asynch support code?)
STAT	T	detailed file info, sorted	STAT	DIR (but not sorted)
STAT	T	report disk space free	STAT	CHKDSK
STAT	T	make drive read-only	STAT	none (can't make drives r/o)
STAT	T	make files read-only, hidden	STAT	none (no file r/o attribute)
STAT	T	verify directory integrity	none	CHKDSK
SUBMIT	T	initiate batch job	SUBMIT	automatic effect of command-search
TOD	T	set/view date, time (see text)	none	DATE, TIME

* These commands are unique to the IBM PC version of the system as supplied by Digital Research.

. . . the XSUB feature of CP/M-80 was omitted from CP/M-86!

the console. Only one batch file may be active at a time. The last line of a batch file may be another SUBMIT command. In this way batch files may be chained, but they cannot be nested.

I was first dismayed, then profoundly disappointed, to find that the XSUB feature of CP/M-80 was omitted from CP/M-86! I cannot comprehend why this *very* useful facility was left out (nor do I understand why MSDOS didn't implement something like it). It should be very easy to create an XSUB for CP/M-86; its absence presents a golden opportunity for software vendors.

The MSDOS command processor invokes a file of batch commands automatically when it cannot turn up a command program. The text-substitution facility of the CP/M-80 SUBMIT utility has been incorporated into the command processor. Operands from the original user command are substituted at marked points in the command lines taken from the file. The user marks these points with "*%n*" rather than with "*$n*" as in CP/M, but otherwise the facility is exactly the same. As in CP/M, batch files can be chained but not nested.

Putting this function in the command processor improves performance, but only slightly. MSDOS avoids having to load a SUBMIT command, and avoids having to write the command file out in internal format. I expected that COMMAND would not have to return to the disk directory before each line, as CP/M does. However, it appears not to be that clever; the disk makes two seeks for every line of the file, just as in CP/M.

MSDOS does not have a facility like XSUB. All requests for console line input must be satisfied from the console. In my opinion, this is an important omission. Many of the MSDOS commands call for stereotyped keyboard responses. Since there is no way to automate these responses, there will be fewer MSDOS batch files that can be left to run unattended.

Object Program Structure

In both systems, new commands are implemented as program files on disk. The systems take different approaches to the preparation of object modules. The 8086 offers many more features for program structure than the 8080, and both systems provide support for them.

8086 Addressing

At any moment, an 8086 object program has access to four different areas of transient storage. The base address of each area is held in a segment register. Storage operands are specified by naming or implying a segment register and giving a byte offset within the segment. The CPU computes the effective address of an operand by adding the offset to the contents of the segment register. The four accessible segments are the Code segment, the Stack segment, the Data segment, and the Extra segment. The segments may overlap. The segment registers can be loaded from storage.

Table 6

MSDOS Command Functions

Command	Type	Function	CP/M-80 equivalent	CP/M-86 equivalent (comment)
(batch)	B	initiate batch execution	SUBMIT	SUBMIT (see text)
COPY	B	general file transfer	PIP	PIP (has more options, stiffer syntax)
ERASE	B	erase directory entries	ERA	ERA
PAUSE	B	pause in batch execution	none	none (but see *DDJ Clinic*, June 1982)
REM	B	remark in batch file	semicolon	semicolon
RENAME	B	rename a file	REN	REN (awkward syntax)
TYPE	B	display file at console	TYPE	TYPE
CHKDSK	T	validate directory integrity	none	STAT
CHKDSK	T	report disk space free	STAT	STAT
CHKDSK	T	report transient space free	none	none
COMP	T	byte-level file compare	none	none
DATE	T	view, set system date	none	TOD
DEBUG	T	debug at machine-code level	DDT	DDT86 (no disk I/O, but assemble command)
DISKCOMP	T	track-level disk compare	none	none (but COPYDISK does a verified copy)
DISKCOPY	T	track-level disk copy	none	COPYDISK
EDLIN	T	line-oriented editor	ED	ED (many more commands)
FORMAT	T	format diskette	(vendor)	(NEWDISK in IBM; vendor in others)
LINK	T	relocating linker	none	none (non-relocating assembler)
MODE	T	set Epson printer options	none	none
SYS	T	copy system programs	SYSGEN	LDCOPY
TIME	T	view, set system time	none	TOD

MASM cannot be loaded. . . with less than 96KB of storage.

It's difficult to avoid confusion in the use of the word "segment" when writing about the 8086. As I have used it so far, a "segment" is an area of storage whose base address is held in a segment register. Such a segment is always 64KB long, because the program can access any offset from 0 to 64KB beyond a segment register. Usually, segments are meant to be much smaller. Although there are no hardware constraints, the programmer will usually think of the Code segment as being only as long as the program code, of the Data segment as being only the size occupied by the defined variables, etc.

Such a logical segment is referred to as a "group" in the documentation of both systems (although neither set of manuals uses very consistent terminology in this area). In an object program, the code group is the concatenation of all code units in the program, the data group is the concatenation of all data units, and so on. In either system, a group may exceed the 64KB that the machine can address at any moment; in that case it is up to the programmer (or compiler) to arrange that the needed part of a group falls within the addressing range of a segment register.

An 8086 program can have a much richer structure than one for the 8080. It may consist of many different groups, and the groups need not be contiguous in storage. On the other hand, an 8086 program may be almost as simple as one for the 8080. If all four segment registers are set to the same base address, the program consists of a single segment of no more than 64KB. If the first 256 bytes of that segment are formatted like the base page of CP/M-80 storage, a very close emulation of the CP/M-80 environment results.

Both operating systems provide a means of putting the parts of a program together into groups in an executable file. Both distinguish between simple one-segment programs and those that have a more complex organization, and both provide an emulation of the CP/M-80 base page.

Program Structure in CP/M-86

CP/M-86 programs are, at present, built with ASM86, the assembler distributed with the system (at the 1982 Computer Faire, Gary Kildall promised that a relocating macro assembler, Pascal, CBASIC, and PL/I would all be available this year). ASM86 is a minimal assembler: it supports IF (but not ELSE), a single-level INCLUDE, and a limited "code-macro" feature that allows the creation of special-purpose op codes (the CompuView system comes with "code-macros" defining the opcodes of the 8087 numerical co-processor).

ASM86 allows code to be assembled into a single code group, and allows the assembly of storage definitions and constants into single data, stack, and extra groups. The groups may be given absolute addresses, or their bases may be left to be defined at load time. No group may exceed 64KB (note that this doesn't restrict the amount of storage that a program can access when running; it limits the amount of code and data that can be defined at assembly time).

ASM86 is not a relocating assembler. This means that programs cannot be assembled as modules to be linked together later. It also means that, except for special-purpose programs that will load at a known address, all programs must be written to be position-independent. This is not difficult to do in the 8086, provided that no group exceeds 64KB. The segment registers are set up on entry to the program; since all operands are addressed relative to the segment registers, it is hard to *avoid* creating position-independent code. Only static inter-group references are position-dependent, because they must contain group base addresses.

ASM86 produces object code in Intel Hex format as an .H86 file. This is made into a loadable .CMD file by the GENCMD utility. A .CMD file contains the binary image of the code, data, stack, and extra groups. It begins with a header record that tells the size of each group and its absolute load address if one was defined. The header also gives a minimum and a maximum allocation size for each group, so that space can be allocated when the group is loaded (the run-time size of a group may exceed its assembled size). Four additional groups can be defined in the .CMD header; these are allocated when the program is loaded, but are not initialized.

If a .CMD file defines only a single group, the program is said to be built on the "8080 Model." Such programs get special treatment when they are loaded.

Program Structure in MSDOS

MSDOS programs are, at present, built with BASIC or with a relocating assembler (other languages are promised from many vendors). BASIC programs are stored in the usual Microsoft tokenized form; they need concern us no further. The MSDOS assembler comes as a separate product ($100 at ComputerLand stores). Two versions are supplied, macro and non-macro. Both produce relocatable object code.

The assemblers are large: a 53KB object file for ASM, the non-macro version, and a 67KB file for MASM. The smaller assembler fills 35% of a diskette. MASM cannot be loaded in a system with less than 96KB of storage. ASM loads in a 64KB machine but produces an "out of memory" error when given a source file much over 100 statements. I had to buy an expansion RAM card before I could assemble useful programs. In my opinion, this kind of bloat is inexcusable. ASM86, which is functionally comparable to ASM, occupies only 27KB. Since an overlarge program is often underfast as well, I timed ASM. When producing only an object file, it assembles 6.7 lines per second; 4 lines/sec when a listing is requested. I did not have a chance to time ASM86, but on my Z80 system MAC runs at nearly 20 lines per second.

The MSDOS assemblers allow any number of "segments" (units of code or data) to be defined and given public names. They allow external references to code entries assembled in other modules. The assemblers produce a .OBJ file of relocatable code. The .OBJ format is said to be compatible with an Intel standard; it is not documented in the IBM manual.

The output of one or more assemblies is combined into loadable form by LINK. LINK seems to be a very competent linker (it's hard to be sure; the IBM manual tries to explain it to novices and ends up saying nothing much to anyone). It organizes the assembled "segments" into groups, of which there can be many. It converts offsets within "segments" into offsets within groups, and corrects static references to group base addresses so they are relative to the start of the program image. It resolves inter-program references. It can apparently search a library of object code to find missing groups, but no documentation is provided on the library format, nor are any library management utilities supplied.

The output of LINK is a .EXE file, a relocatable program

ASM86 is not a relocating assembler.

image. The image may contain static references to group base addresses, which must be relocated to account for the load address of the program.

The Execution Environment

Both systems contain a means of loading a program and setting up its execution environment: its segment registers, its allocated storage, and its means of communication with the rest of the system. In both systems, the execution environment is centered on a 256-byte base page.

The CP/M-86 Environment

The CP/M-86 BDOS contains the logic required to load a .CMD file. The file header describes the groups that make up the program. The BDOS allocates storage for each group, and then loads the program image into its various group-areas. All the groups are put into a contiguous area if one exists, but if not, the BDOS will put each group where it fits. Only four groups are loaded from the file, but the file can specify up to four additional areas to be allocated for working storage. The size of any group will have been set by GENCMD from its command operands, and may be from 16 bytes up to 1MB.

The BDOS does not relocate static base addresses in the code. It does place the base address and length of each allocated group within the base page, which occupies the first 256 bytes of the Data group. The length of the Data group falls at offset 0006 in the base page.

The BDOS load function is available to any program, but its most frequent user is the CCP. After the BDOS has loaded a program, the CCP sets up the command operands in its base page in the CP/M-80 format. It sets the segment registers and enters the program. A program on the "8080 model," with a single group, has all its segment registers set equal, so as to simulate the 8080 environment. Other programs receive their segment registers set to the base of the four basic groups; to address other groups, or to access beyond 64K in a group, the program must load its own segment registers via the base page.

A program communicates with the BDOS by loading a function number into the CL register, an address into the DX register, and issuing an INT 224 (*not* by calling a vector in the base page; except for single-group programs, the base page is not part of the Code segment and so can't be called). The BIOS can be called by requesting a BDOS function. The BIOS functions are identical to those of CP/M-80.

A program terminates by requesting BDOS function 0 (*not* by jumping to the base page offset 0000). Two kinds of termination are allowed. Normally a terminating program is removed; all its allocated storage is released. But a program may request that it not be removed from storage. Such a program loses control of the CPU, except as it is reentered by an interrupt. Ordinary programs have no use for this feature, but it offers a number of opportunities to systems programmers. The 8086 has 256 different interrupt codes, most of them available to software calls via the INT instruction. A device driver, a spooler, an XSUB, a numerical processor, or any other kind of software resource could be built with this mechanism. It would load as a command, and stay around permanently after "terminating" to service its particular interrupts. MSDOS has a similar feature, as we'll see.

The MSDOS Environment

In MSDOS the program load function is entirely in the COMMAND program. MSDOS distinguishes two kinds of loadable files, .COM and .EXE. The former are single-group, non-relocatable programs – programs built on the 8080 model. They are loaded without relocation (so must be position-independent), and are entered with all segment registers pointing to the base page. The IBM system provides no command for creating a .COM program (one could be built with the DEBUG program, but this is not specifically documented). A .COM program can be loaded by another program and then called.

When a .EXE program is loaded, header information in the file guides COMMAND in relocating all static group-base addresses to account for the load address of the program. The entire program image is loaded contiguously at the lowest available address (there is a LINK option that will cause COMMAND to load one group of the program at the highest possible address). On entry to a .EXE program, the Code segment register bases the first code group and the Data segment register addresses the first data group. Other groups can be accessed by loading the segment registers with immediate operands which have been relocated. The .EXE file format is not documented, so a .EXE program can't be loaded by another program.

The CP/M-like base page is addressed through the Extra segment register. In it, the size of storage is available at offset 0006, and the command operands are available in the usual places. I found two small mistakes in this emulated base page. The command tail at offset 0081h is not translated to upper case as was done in CP/M-80. This could be regarded as improvement, but it is a change. And the last character in the tail is a CR character, not an 00h byte.

Once loaded, a program communicates with the BDOS by loading a function number into the AH register, an offset into the DX register, and issuing an INT 33 (a .COM program *may* CALL 0005h instead; there is a "far call" to the BDOS at that address). The BIOS can be called by issuing one of several interrupt codes. However, not only are these BIOS functions quite different from those of CP/M-80, they will probably differ from vendor to vendor. Use of the BIOS should be avoided in MSDOS (as it should be avoided in CP/M-86 as well, but for other reasons).

The program terminates by setting the Code segment register to address the base page and issuing an INT 32 (it *may* jump to offset 0000h in the base page; there is a INT 32 there). The program has the option of setting DX to its end-offset and issuing an INT 39; this returns control to MSDOS but the program will be preserved in storage where it may respond to hardware and software interrupts. MSDOS adjusts its program load point upward to avoid overlaying the program.

The MSDOS environment has one unique feature: it is possible for a program to trap both control-break interrupts and permanent disk errors. The program can defeat a user's attempt to abort it, or can provide a graceful termination when control-break is hit. It can gain control when a disk error occurs and either take some action or notify the user in terms relevant to the application. Both of these features strike me as useful. They are available only at the assembly-language level, of course.

The BDOS Services

Table 7 summarizes the BDOS functions available in the two systems as they compare to their model, the CP/M-80 functions. Certain functions have been marked as having a "significant" incompatibility with CP/M-80. By "significant" I mean that *some* programs will require logic changes to operate with these BDOS requests. I think that only a minority of programs will be affected, and the required changes will be minor – but study Table 7 and form your own opinion. The MSDOS file services will be discussed later. The only real difficulty that I see is that MSDOS does not provide function 12, Get Version Number. This seems a silly oversight; if MSDOS supported it (returning some unique value) then we could write programs that would run in either system. (On second thought, maybe that wasn't an oversight.)

Function 11 is an interesting case. The CP/M-80 manual claimed that if a byte was available, register A would be returned with FFh. In fact, CP/M-80 returned 01h. The MSDOS designers chose to implement the CP/M-80 interface *as documented*. The CP/M-86 designers implemented the CP/M-86 interface *as it was*, but documented it accurately. Which is more "compatible"?

MSDOS function 41, Parse Filespec Into FCB, is a welcome addition; it should be a great convenience to programs that take a filespec from a user input. It is also nice to find services 42-44 for operating on a system clock and calendar without hardware dependency.

CP/M-86 functions 53-58 mark a major enhancement over CP/M-80. The BDOS controls all storage allocation. If a program needs more storage, it can call on a variety of BDOS functions to allocate or release space. This, in my opinion, is an important feature. Since the BDOS manages storage allocation, programs are protected from each other. A program can acquire the storage it needs without any consideration of the presence of other programs. Storage can be acquired in blocks of any size, and the blocks need to be contiguous – the BDOS will allocate around any hardware holes and the program need not be aware of them. This feature is related to the program-load BDOS function. The calling program does not have to sub-allocate its own space to hold the loaded program; the BDOS allocates whatever space the loaded program needs. In MSDOS, storage allocation is handled in the same *ad hoc* way that it was in CP/M-80 (with the single exception that an interrupt server is protected from programs loaded after it).

The File Systems

The file system is the heart of any operating system. The CP/M-86 system is quickly dealt with: it is identical to that of CP/M-80. FCBs, directory entries, the Disk Parameter Block, the Allocation Vector – all are identical in layout and use. Your un-erasers, disk zappers, and catalog programs will need no logic changes when ported to CP/M-86.

Nor will they find any improved function. This disappoints me. MP/M II and MP/M-86 both contain major improvements in the file system. They support file passwords, multirecord transfers, creation and access timestamps, error trapping, and disk label identifiers. I don't see why these things weren't made available in CP/M-86.

The MSDOS designers attempted to improve on the CP/M-80 file system while maintaining compatibility for the majority of programs. The changes they made fall into three categories: file access to serial devices, record I/O, and a different method of allocating disk space. (The information that follows is based on the IBM DOS manual only; I did not have time to verify it by experiment. I found IBM's manual a skimpy, cryptic, and ambiguous guide in these matters. Some of what follows is therefore based on inference and extrapolation which I believe to be sound but which may be wrong in detail.)

File Access to Serial Devices

If one of the device names shown in Table 3b is used as a filename in an FCB, the BDOS will direct all operations on that FCB, not to a disk file, but to the device named. If the FCB filename is "CON:" then a request for a sequential read will bring keyboard data to the file buffer while a request for sequential output will write to the screen. This seems to me a most useful concept. For example, it allows a utility like COPY to transfer data from a file to the printer without special programming and without special command syntax. This flexibility is available to any utility that takes a filespec in its command operands.

The feature does not, in my opinion, replace the IOBYTE feature of CP/M-86; it is a different thing entirely. In MSDOS the user can cause a file-access program to use a serial device instead. In CP/M-86, the user can cause a serial access program to use a different serial device. In both cases the redirection is transparent to the program. In MSDOS, the user cannot reassign printer output to a different port. In CP/M-86 the user cannot make a disk utility write to the printer instead. And in neither system can the user make a program that uses the serial I/O functions use disk data instead.

Record I/O

MSDOS eliminated the CP/M-80 restriction that all file access must be in units of 128 bytes (do keep in mind that this restriction affected only those who programmed in assembly language; all but the simplest language translator masked it). The BDOS will transfer a record of any size, and it will transfer a block of many records (up to 64KB total) on a single request.

In order to access a file in units of, say, 439 bytes, you would set that record size in a field of the FCB after opening the file. When the FCB is first opened, the record size is initialized to 128 bytes. As long as it is left so, the file functions will behave like the CP/M-80 services. The current-record and random-address fields will have the same meanings.

If a file is written in 128-byte units, it can be read back in 128-byte units without difficulty. If a file is written in units that are not multiples of 128, and is then read back in 128-byte units, the last record may be a short one (that is true whenever the writing and reading units differ). The BDOS returns a code of 01h to signal end of file, and a code of 03h to signal a short last record. A program ported from CP/M-80 will treat both codes as end of file. That is why function 20 is marked as incompatible in Table 7.

Your un-erasers, disk zappers, and catalog programs will need no logic changes when ported to CP/M-86.

If the record size is set to 1, function 35 will return the file size in bytes. Functions 39 and 40 allow the program to read or write as much as 64K bytes in one operation, a convenience for some applications.

The fact that MSDOS will transfer data in other units than 128 bytes does not necessarily make its I/O more efficient than that of CP/M-86. The CP/M-80 record size is an artifact of its BDOS-to-program interface. Many CP/M-80 systems, and most CP/M-86 systems, transfer data between computer and disk drive in larger units. My Z80 system, for example, usually transfers data in 1024-byte sectors. Some systems read and write whole tracks; others, like the Intersystems "Cache BIOS" machines, buffer multiple tracks in storage. The transfer blocksize, channel bandwidth, and disk seek time govern the I/O performance, not the record size at the BDOS interface. By using the MSDOS multirecord read and write functions the programmer can ensure that it won't take more than two revolutions to write a track's worth of data – an effect that isn't difficult to achieve under CP/M-80, either.

Disk Space Allocation

The MSDOS method of space allocation is completely different from that of CP/M-80 (see box below). Disk space is allocated in units which the IBM manual calls "sectors" (I'm not sure that these units now, or will always, correspond to physical sectors; hence the quotes). Each sector is 512 bytes long. There are 313 sectors on a diskette in the IBM system, of which the first 8 are reserved for system purposes. The rest are numbered from 2 up to 304. A File Allocation Table (FAT) is kept in a reserved sector. It is a vector of 305, 12-bit entries. Each entry represents the status of the corresponding sector, with an entry of 000h standing for a free sector and FFFh meaning the last sector of a file.

MSDOS maintains a directory of 32-byte entries, just as CP/M does. However, an MSDOS entry contains no data map; it holds only the number of the first sector assigned to the file. Call that number F; then the number of the following sector is in FAT[F], and so on through the FAT to the last sector of the file, for which FAT[F] =FFFh. In order to allocate a sector, the BDOS finds the first FAT[N] =000h, and adds sector N to the file. Duplicate copies of the FAT are stored in reserved sectors; a copy of the FAT is kept in storage at all times.

Because the FAT is in storage, the BDOS can always find the number of the next sector of a file without reading the directory. Because most allocation data is in the FAT, a file never requires more than one directory entry, and that entry has room for a creation date and a file size. An entry also has room for the eleven file attribute bits supported by CP/M and MP/M, but MSDOS does not implement these. An MSDOS file can be "hidden," equivalent to the SYS file attribute, but not by any command available to the user. So far as I could determine, the read-only, archive, and application-defined attributes of CP/M-80 are not supported. Because the directory entry differs from CP/M-80, any CP/M-80 program that looks at the attributes, extent number, record count, or data map of an FCB will have to be redesigned for MSDOS.

It is interesting to calculate maxima for the MSDOS allocation scheme. A FAT element is a 12-bit integer. There can be at most 4,093 unique sector numbers (002h-FFEh). A maximum FAT would occupy 6,144 bytes in storage and 12,288 bytes on disk (two copies). Such a FAT could track the allocation of 4,093 sectors, or 2,095,616 bytes of data if a sector is 512 bytes. Unless some parameter changes, *the largest possible MSDOS file will contain only 2MB of data*, one-fourth the maximum size allowed by CP/M-86!

This is strange, for the record-size field in an MSDOS directory entry is a 4-byte integer. In order to allocate a file of 2^{31} (2.1 billion) bytes, MSDOS would have to use either a FAT with 4,194,304 elements or a sector size of 524,672 bytes. Or it could compromise on a FAT of 65,533 entries tracking "sectors" of 32KB each.

Mind you, the IBM manual says nothing to suggest that the FAT could have more than 305 entries, or that a FAT element might be other than 12 bits wide, or that a "sector" would ever be other than 512 bytes. It is my inference that one or more of these things must be a system parameter. For instance, to control a double-sided, double-density, 8-inch diskette (1.26MB), the FAT must have 2,464 entries. To control a 5MB hard disk, either the FAT entries must be 16 bits or the sector size must rise to 2 KB.

For readers who don't know how CP/M-80 allocates space, here is a summary. CP/M divides a disk into a number of blocks B, each of a size S. The blocks are numbered from 0 to B-1. The first D blocks are assigned to be a directory. On a single-density 8-inch diskette, B=243, S=1024, and D=2. The numbers vary with the disk hardware; they are chosen by the author of the BIOS.

A file is represented by one or more directory entries. An entry contains a data map, which is simply a list of the numbers of the blocks that hold data for that file. One data map accounts for one *extent* of data – usually 16KB, but the size depends on B and S. A file that exceeds one extent has more than one directory entry; each entry has a unique extent number so the file can be retrieved in order. When the program reads past an extent boundary the BDOS must return to the directory for another data map.

When it first accesses a disk, CP/M builds a vector of B bits, setting the first D bits to 1 and the rest to 0. Then it reads all directory entries. For every block number in a data map, the corresponding bit is set to 1. The zero bits that are left represent free blocks; the index of the first 0-bit is the number of the first free block. That number is put in the data map of the next file that needs a block of space.

Table 7

nbr	CP/M-80 Function	MSDOS Differences	CP/M-86 Differences
0	terminate program	CS must address base-page prefix	*DL must contain a parameter
1	CON: byte input, tab expand., no ^C check	tabs returned, ctl-break tested	
2	CON: byte output, tabs expanded	*tabs written, bkspce is destructive	
3	RDR: byte input	input from asynch port #1	input from AXI: (name change only)
4	PUN: byte output	output to asynch port #1	output to AXO: (name change only)
5	LST: byte output		
6	CON: unedited output/input without echo		also supports kbd-status test
7	get IOBYTE	*direct console input	
8	set IOBYTE	*direct console input	
9	CON: string$ output, tabs expanded	tabs not expanded	
10	CON: line input with editing controls	*returned line ends in CR, not 00h	
11	CON: input status, returns 00/01	returns FF/00 (see text)	returns 01/00 (see text)
12	get system, version numbers	*clear template and do kbd input	
13	reset all drives	*also resets file transfer address	
14	select a drive	also returns number of drives	
15	open (an extent of) a file	*extent number not relevant	
16	close (last extent of) a file		
17	search for first matching FCB	*full-search option not supported	
18	search for next matching FCB	*full-search option not supported	
19	delete file		
20	sequential read of 128 bytes	*partial-record return code possible	
21	sequential write of 128 bytes	other record sizes supported	
22	create (first extent of) a file	system prevents duplicate filenames	
23	rename file	"?" in new name copies old name byte	
24	get bit map of active drives	*not supported	
25	get default drive's number		
26	set file transfer address	sets both segment base and offset	sets offset only; see #51
27	get address of disk allocation vector	returns address of F.A.T.	
28	make the current drive read-only	*not supported	
29	get bit map of read-only drives	*not supported	
30	set file attributes	*not supported	
31	get address of disk parameter block	*not supported	
32	get/set user (sublibrary) number	*not supported	
33	direct-access read of 128 bytes	other record sizes supported	
34	direct-access write of 128 bytes	other record sizes supported	
35	get size of file in 128-byte units		
36	get direct-access file position		
37	reset selected drives	*set an 8086 interrupt vector	
38	n.a.	create a program base page	n.a.
39	n.a.	direct read multiple records	n.a.
40	as #34, but fill new blocks with zero	*direct write multiple records	
41	n.a.	parse filespec string into FCB	n.a.
42	n.a.	return system date	n.a.
43	n.a.	set system date	n.a.
44	n.a.	return system time	n.a.
45	n.a.	set system time	n.a.
50	n.a.	n.a.	call BIOS function
51	n.a.	n.a.	set file tranfer segment base
52	n.a.	n.a.	get file transfer base and offset
53-58	inclusive: n.a.	n.a.	calls to get, free transient storage
59	n.a.	n.a.	load a program from a .CMD file

*a significant difference from CP/M-80 function which might require logic changes in more sophisticated programs.

. . . claims that it could cope with files of 2^{31} bytes are, in my opinion, absurd.

I draw three conclusions from these calculations. First, as distributed and documented by IBM, MSDOS has a maximum file (and disk) size of 2MB, less than that of CP/M-86. Second, it could be modified to support much larger files, but the interface to do that is not documented (unlike the BDOS-BIOS interface of CP/M-86). Third, MSDOS cannot practically support files very much larger than those of CP/M-86; claims that it could cope with files of 2^{31} bytes are, in my opinion, absurd.

Table 8

Extensions to CP/M-80 That Do NOT Appear

Volume label support; volume-id verification; use of volume-id in place of a drive-letter in commands.

Multi-volume disk files.

System-supported archival storage; catalog of archived files; archive command to back up only changed files.

Clock-based job scheduling; commands executed at preselected times and dates (such as archive backup runs).

Multiple directories; cross-volume directories; directory as a standard filetype with creation and update utilities; archival backup of groups of files as named in a subdirectory.

True I/O redirection (transparent redirection of serial I/O requests to a disk); serial output redirection to another program's input; DOS-supported spooling of serial I/O; asynchronous printing scheduled without constant user intervention.

BDOS functions for device-independent control of cursor and screen; BDOS function to return screen dimensions (1 x 80 when screen output redirected).

Support of (1) default operand values, (2) conditional tests, (3) iteration, in batch command language; nested batch files.

Table 9

Significant Regressions from CP/M-80 Function

CP/M-86	MSDOS
	no serial I/O redirection
	no command to make a file r/o
XSUB facility omitted!!!	media incompatible with other systems
	no archive attribute in files
	no XSUB facility

Summary

Thus far in this paper I have laid the systems out for you side-by-side. Now let's look at them head-to-head. I invite you to participate in this: feel free to add or delete entries from the tables I'll present, or even to switch entries from one column to another.

CP/M-80 was a sturdy little tyke, but by no means was it a complete operating system. After examining these descendants of CP/M-80, my strongest feeling is one of disappointment. It disappoints me that two high-powered software houses have worked so hard to produce only . . . two more CP/Ms. Consider Table 8, a very incomplete list of things, all needed, all feasible, that do not appear in either system.

Now consider Table 9, which summarizes the points in each system that (I think) represent significant *regressions* from the capabilities of CP/M-80.

For a more cheerful view, see Table 10, which lists what I feel are the significant improvements on CP/M-80. It only bothers me that the two columns of Table 10 are so different from each other.

Finally, let us compare the systems directly. Table 11 lists what I feel are the significant advantages of CP/M-86 over MSDOS – *wait for it!* – while Table 12 lists what I see as the significant advantages MSDOS has over CP/M-86. Ok, now look.

Table 10

Significant Improvements on CP/M-80 Functions

CP/M-86	MSDOS
Documentation vastly improved.	Command documentation clear, handsomely presented.
Searches for commands under user code zero.	Automatic batch execution.
No I/O on warm start; A-disk need not be bootable.	No I/O on a warm start; A-disk need not be bootable.
DIRS to display system files.	Liberal rules for disk change.
HELP command provided.	Consistent support for 1-drive systems; COPY is a built-in command.
PIP copies to other user codes.	RENAME syntax much better.
STAT validates the directory.	Print-screen function (may be specific to IBM hardware).
Coherent storage-management scheme handled by BDOS.	Multi-record I/O requests.
Easy to implement interrupt-servers.	Serial device may be treated as files through FCB.

The Last Word

I am not going to present a recommendation as to which system you should prefer. This has been a technical comparison. Your choice of an operating system will be determined by a lot of considerations, only a few of them technical. I'm not even going to tell you which I would prefer to have in my own machine, because I don't want that taken as a recommendation by someone in different circumstances than mine.

I will make these recommendations. Software developers: these systems are similar enough that your products can run on both with only minor changes. Modularize your programs so as to encyst every system call in a subroutine of your own. Include those units from a library (cystem.lib?). Then with a single library change you can port between systems, and you will not be dependent on the choices of the OEMs that built the systems. Hackers and homebrewers: consider which system is the most up-front with information on its internals. That's the one you can most easily adapt to your peculiar hardware.

And to ordinary users: you really don't care, do you? You should be looking at the applications that are available. Choose your applications first, then get the system that runs all of them. If possible, wait at least another twelve months; the applications market for all 8086-based systems is just about to explode. ▶▶J

Table 11

Significant Advantages of CP/M-86 Over MSDOS

1. I/O redirection through IOBYTE.
2. R/O protection for files and drives, via user commands.
3. File-transfer utility much more capable (PIP vs. COPY).
4. Consistent, extensible scheme of file attributes.
5. More options for directory display (DIR, DIRS, STAT).
6. Coherent storage-management scheme built into system.
7. Assembler (non-macro, non-relocating) supplied with system; compiles sizable programs in 64KB machine.
8. 8-inch diskettes interchangeable with CP/M-80, MP/M-80, MP/M-86 systems and some others; 5-inch diskettes interchangeable with some systems, hardware permitting.
9. Interrupt servers (spoolers, device drivers, XSUB?) easy to implement.
10. Adaptable to all disk hardware via well-documented interface.
11. Upgrade path to MP/M-86 available now.

Table 12

Significant Advantages of MSDOS Over CP/M-86

1. Automatic invocation of batch command files.
2. Liberal rules for changing diskettes.
3. File-transfer utility is a built-in command.
4. Consistent support for single-drive systems.
5. Prior command can be recalled, re-entered.
6. Serial device names can be used as command operands where the program expects a filespec.
7. Full relocating linker supplied with system; system contains a relocating loader.
8. File system supports various record sizes and multi-record access requests.
9. Lower overhead on random file access.
10. Control-break and disk error trapping.
11. BDOS functions for system clock/calendar.
12. Upgrade path to XENIX promised.

Graphics on IBM's Personal Computer

by Ray Duncan

The IBM Personal Computer is available with two types of video output: a monochrome 80 column by 25 line text-only display, and a color/graphics display with text and bit-mapped graphics capabilities. Both are memory mapped; software drives the displays by simply writing character codes or bit patterns directly into a designated area of RAM. The memory is dual ported and is accessed "from the back" by the Motorola 6845 chip which controls the monitor.

Referring to the memory map in Figure 1, you will notice that the monochrome display occupies 4Kbytes starting at B0000H, and the color/graphics display is assigned 16Kbytes starting at B8000H. In addition, IBM has reserved a total of 112 Kbytes of address space as "graphics/ display buffer" which implies plans for release of video displays with much higher resolution at some later date.

The discussion in this article will be confined to use of the color/graphics interface in high resolution text and graphics modes. Although the version of BASIC supplied with the Personal Computer contains powerful high-level commands for both text formatting and creation of graphic images, this is hardly a suitable language for the many utilities, editors, and games that all of you ambitious programmers are thinking about.

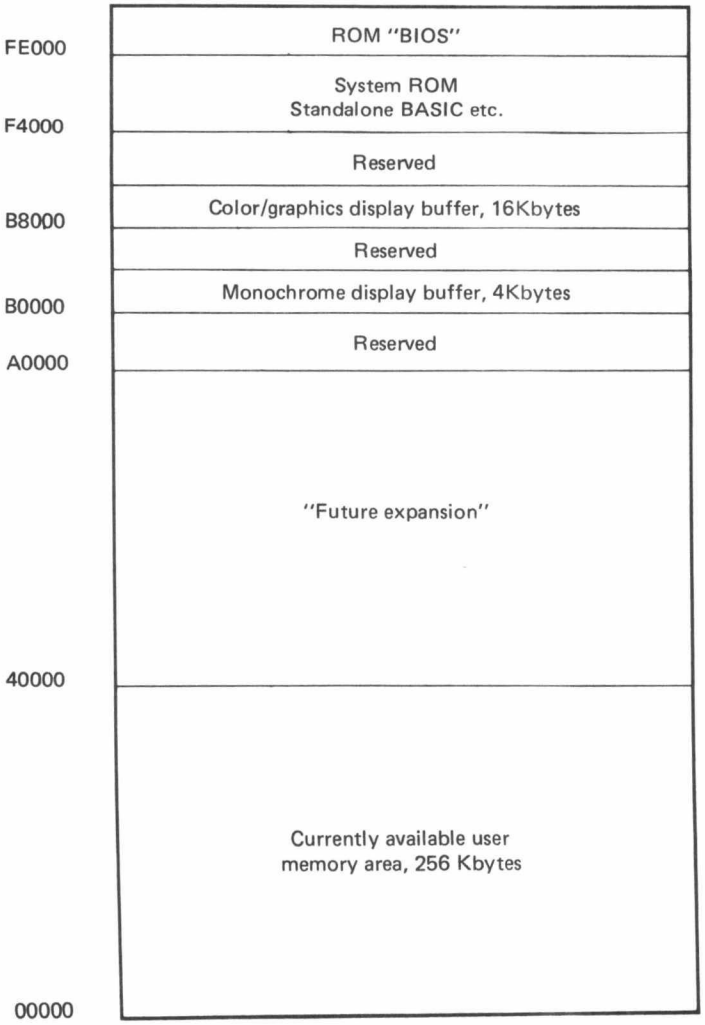


Figure 1
Memory map of the IBM Personal Computer.

Color/graphics Board High-Res Text Mode

In this mode an 80 column by 25 line text area is displayed. Characters are 5 by 7 dot patterns defined in an 8 by 8 box with one line of descender for lower case. The cursor home position, (X,Y)=(0,0), is considered to be the upper left corner of the screen (see Figure 2).

Each character display position is allotted two bytes in the RAM buffer. The first byte (even address) is the ASCII code of the character, which is translated by a character generator into a dot matrix pattern for the screen. The contents of the odd address is called the attribute byte. Several bit fields in this byte control such features as blinking, highlighting, and reverse video (see Figure 4).

A hex and ASCII dump of part of the video map in text mode is shown in Figure 5. Since each screenful of text takes (80 columns * 25 lines * 2 bytes per character) = 4000 bytes, and the color/graphics interface has 16Kbytes of on-board memory, the video map is divided into 4 Kbyte "pages." The portion of memory currently being displayed is called the "active page." While the user is viewing one portion of text, a program can be building up a new text image in another part of the buffer and then cause a new image to appear instantaneously by simply selecting a new active page.

The memory address of any character in the display may be calculated as the base address of the RAM buffer, plus the current active page number times the size of the page, plus the line number (Y coordinate) times 80 characters per line times 2 bytes per character, plus the column number (X coordinate) times 2 bytes per character, i.e.:

address = B8000H + (page * 1000H) + ((Y * 50H + X) * 2)

High-Res Graphics

There are two levels of graphics available to the programmer: 320 x 200 black and white or color and 640 x 200 black and white. Only the latter has sufficient resolution for any sophisticated graphing and design work, although the color capabilities will undoubtedly be very popular for sales demonstrations, games, and flashy business programs.

In high resolution black-and-white mode, each bit of the memory map is a "pixel" which is either on or off and corresponds directly to a dot on the screen. When addressing the display, the "X" coordinate giving the horizontal displacement must be in the range 0 to 639, and the "Y" coordinate giving the vertical displacement must be in the range 0 to 199. The home position, (X,Y)=(0,0), is the upper left corner of the screen (see Figure 3).

Translating an (X,Y) coordinate into a physical memory address is more complex in graphics mode than in text mode. Each successive 80 bytes, or 640 bits, consists of one horizontal scan line. Within each byte, the bits map one-for-one onto points, with the most significant bit corresponding to the leftmost displayed dot of a set of 8 dots, and the least significant bit corresponding to the rightmost displayed dot of a set

of 8 dots. The memory map is set up so all the even Y coordinates are scanned as a set alternating with all of the odd Y coordinates.

The byte address containing a particular (X,Y) coordinate may thus be calculated as:

address = B8000H + ((Y AND 1) * 2000H) +
(Y/2 * 50H) + (X/8)

Once the correct byte is extracted, the bit number may be calculated as:

bit number = 7 - (X MOD 8)

where bit #7 is the most significant bit, and bit #0 is the least significant bit. It is easiest to build an eight-byte table or array of bit masks, and use (X AND 7) to extract the appropriate entry from the table.

IBM Video Drivers

A complete set of text and graphics drivers are present as part of the machine's ROM software. These routines are accessed through a software interrupt trap and provide display mode selection, cursor addressing, display of text, scrolling, and point plotting. Documentation of these services is virtually nonexistent in the IBM manuals except for the ROM BIOS program listing itself in the back of the Technical Reference.

The most elegant feature of the video support routines is that most of them may be called by user software without regard for the display controller or video mode in use. The subroutines determine the current type of display and perform all necessary address translation, also fetching bit patterns for characters from a resident table if the program wishes to display text in graphics mode. Naturally one must pay for this flexibility in speed; the user can readily write much faster drivers for dedicated application programs where the video display mode is known in advance.

The common calling format for the video routines is as follows:

```
MOV  AH,function-number  ; AH always determines call type
  .                      ; load other
  .                      ; registers with
  .                      ; call-specific parameters
INT  10H                 ; trap to service routine
```

Although each function call has different arguments and returns different results, some general rules can be applied:

- The BX, CX, and DX registers and all of the segment registers are preserved across video driver calls; the contents of all other registers may be destroyed.
- The X coordinate or column number is passed in register CX or DL.
- The Y coordinate or row number is passed in register DX or DH.
- The display page, if any, is passed in register BH.
- The character or pixel value to be written is usually passed in register AL.

Details of Video Driver Services

Function 0 – set video mode

CALL with:
AH = 0
AL = 0 40 x 25 black and white text
1 40 x 25 color text
2 80 x 25 black and white text
3 80 x 25 color text
4 320 x 200 color graphics
5 320 x 200 black and white graphics
6 640 x 200 black and white graphics
7 monochrome text display

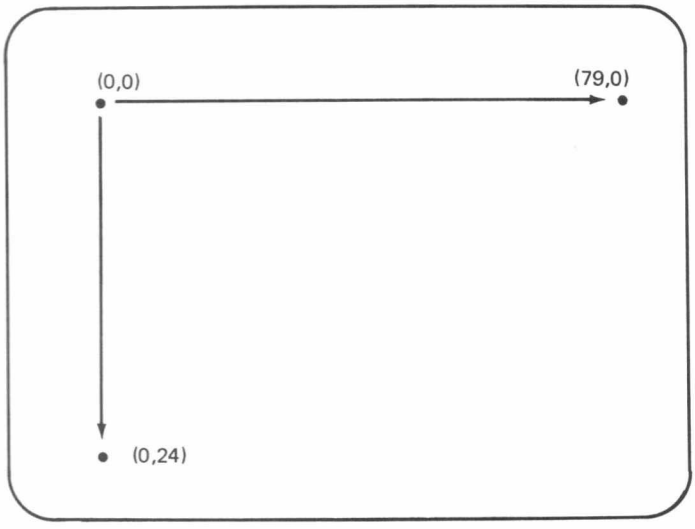


Figure 2

Cursor addressing, color/graphics interface in high-resolution text mode (mode 3).

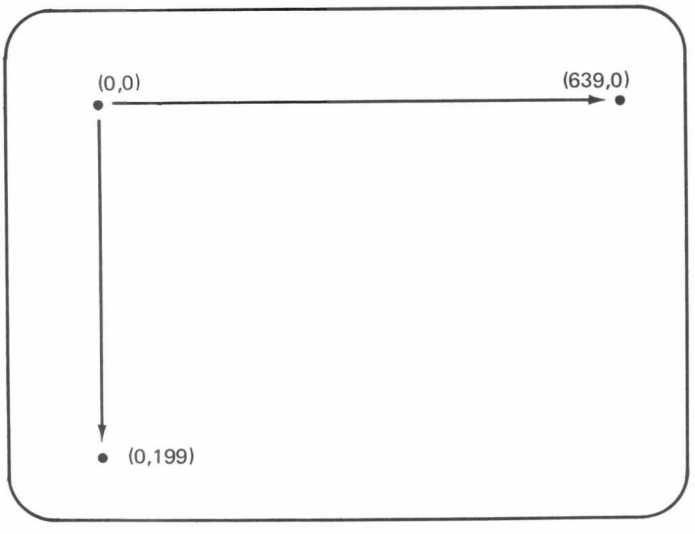


Figure 3

Point addressing, color/graphics interface in high-resolution graphics mode (mode 6). Black and white display, 640 x 200 resolution.

(Continued from page 33)

Function 1 – set cursor type

Call with:
AH = 1
CH = bits 0-4 starting line for cursor
CL = bits 0-4 ending line for cursor

Note: Hardware will always cause blink. Setting bit 5 or 6 in CH will have unpredictable effect on cursor.

Function 2 – set cursor position

Call with:
AH = 2
BH = page number (must be zero for graphics modes)
DH = row (Y coordinate)
DL = column (X coordinate)

Note: (X,Y)=(0,0) is upper left corner of screen.

Function 3 – read cursor position

Call with:
AH = 3
BH = page number (must be zero for graphics modes)
Returns:
DH = row (Y coordinate)
DL = column (X coordinate)
CH = starting line for cursor
CL = ending line for cursor

Function 4 – read light pen position

Call with:
AH = 4
Returns:
AH = 0 if light pen not down / not triggered
1 if light pen down and triggered
DH = character row (Y coordinate, 0-24)
DL = character column (X coordinate, 0-79)
CH = pixel row (Y coordinate, 0-199)
BX = pixel column (X coordinate, 0-319 or 0-639 depending on mode)

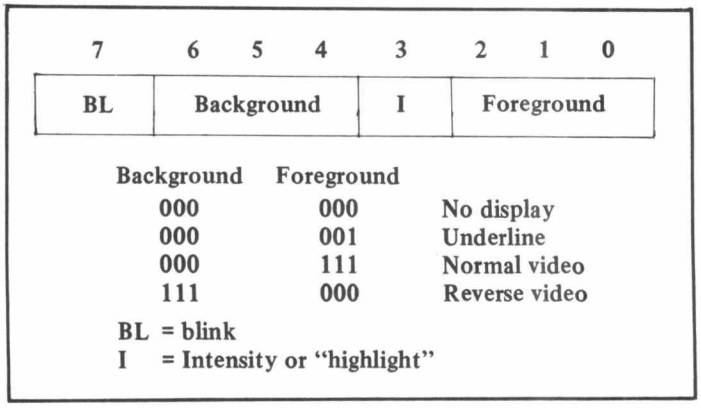


Figure 4

Attribute byte for each displayable character in text mode. Blink, intensity, and foreground/background attributes may be used in any combination. In reverse video, the intensity bit has the opposite effect from what you would expect: the contrast of the character is halved instead of doubled.

Function 5 – select active display page

Call with:
AH = 5
AL = page value

Note: This function is legal only in text display modes. The page number may be in the range 0-7 for modes 0 to 1, in the range 0-3 for modes 2 and 3.

Function 6 – scroll window of active page up

Call with:
AH = 6
AL = number of lines to scroll, new lines are blanked at bottom of window. If AL=zero, entire window is blanked.
CH = Y coordinate, upper left corner of window
CL = X coordinate, upper left corner of window
DH = Y coordinate, lower right corner of window
DL = X coordinate, lower right corner of window
BH = attribute to be used for blanked area.

Note: Function 6 and 7 have tremendous potential for support of object-oriented editors and languages.

Function 7 – scroll window of active page down

Call with:
AH = 7
AL = number of lines to scroll, new lines are blanked at top of window. If AL=zero, entire window is blanked.
CH = Y coordinate, upper left corner of window
CL = X coordinate, upper left corner of window
DH = Y coordinate, lower right corner of window
DL = X coordinate, lower right corner of window
BH = attribute to be used for blanked area.

Function 8 – read attribute and character at cursor

Call with:
AH = 8
BH = display page
Returns:
AL = character
AH = attribute byte (text modes only)

Function 9 – write attribute and character(s) at cursor

Call with:
AH = 9
AL = character to write
BH = display page
BL = attribute
CX = count of characters to write

Notes for functions 9 and 10: writing multiple characters by setting the replication factor in CX will only produce a valid result for the current row. All values of AL result in some sort

of text or graphic character appearing on the screen; i.e., the codes normally thought of as line feeds or carriage returns do not have any effect on the cursor. After a character is written, the cursor must be moved explicitly to the next desired position using function 2. The user is responsible for detecting end-of-row or end-of-screen and for taking the appropriate action.

Function 10 – write character only at cursor

Call with:
AH = 10
AL = character
BH = display page
CX = count of characters to write

Function 11 – set color palette

Call with:
AH = 11
BH = color palette ID being set (0-127)
BL = color value to be used with that color ID

Note: Color ID=0 selects the background color (0-15 in graphics mode, and 0-31 in text mode).

Function 12 – write dot

Call with:
AH = 12
AL = color value. If bit 7 of AL is set, the color value is exclusive OR'd with the current contents of the pixel.
DX = row number (Y coordinate)
CX = column number (X coordinate)

Function 13 – read dot

Call with:
AH = 13
DX = row number (Y coordinate)
CX = column number (X coordinate)
Returns:
AL = pixel contents

Function 14 – write text in teletype mode

Call with:
AH = 14
AL = character
BH = display page in alpha mode
BL = foreground color in graphics mode

Note: If the character is line feed, carriage return, or backspace, the appropriate cursor movement is executed. If the character is a bell code, a tone is sounded. Otherwise, the character is placed on the display at the current cursor position and the cursor is moved to the next position. If the cursor is at end-of-line, it is moved to the start of the next line. If the cursor is on the last line of the screen, the screen is scrolled up and the cursor is placed on a new line. Beware! When scrolling, the entire new line is set to the attribute of the last character that was written on the preceding line. This can produce some alarming and visually rather distasteful results when using reverse video.

Function 15 – get current video state

Call with:
AH = 15
Returns:
AL = mode (see function 0 for explanation)
AH = number of character columns on screen
BH = active display page

A Line Generator

To further the cause of independence from BASIC for graphics support, I am including two logically equivalent

(Continued on page 36)

```
          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
B800:0000 54 07 68 07 69 07 73 07 20 07 64 07 65 07 6D 07
B800:0010 6F 07 6E 07 73 07 74 07 72 07 61 07 74 07 65 07
B800:0020 73 07 20 07 74 07 68 07 65 07 20 07 63 07 6F 07
B800:0030 6E 07 74 07 65 07 6E 07 74 07 73 07 20 07 6F 07
B800:0040 66 07 20 07 74 07 68 07 65 07 20 07 63 07 6F 07
B800:0050 6C 07 6F 07 72 07 2F 07 67 07 72 07 61 07 70 07
B800:0060 68 07 69 07 63 07 73 07 20 07 20 07 20 07 20 07
B800:0070 20 07 20 07 20 07 20 07 20 07 20 07 20 07 20 07
B800:0080 20 07 20 07 20 07 20 07 20 07 20 07 20 07 20 07
B800:0090 20 07 20 07 20 07 20 07 20 07 20 07 20 07 20 07
```

Figure 5

Dump of first 160 bytes of color/graphics interface memory buffer in 80 x 24 text mode. Note that ASCII character codes are stored in even bytes and their respective attributes in odd bytes.

```
          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
B800:0000 FF FF FF FF FF FF FF F8 00 00 00 00 00 78 FC C6
B800:0010 00 FC 00 00 00 00 00 00 70 00 3C 00 00 00 00 10
B800:0020 00 00 00 00 00 08 00 00 00 00 00 00 00 00 00 00
B800:0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
B800:0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
B800:0050 80 00 00 00 00 00 00 08 00 00 00 00 00 30 66 FE
B800:0060 00 66 78 DC 7C 78 F8 78 30 00 C0 78 CC DC CC 7C
B800:0070 78 DC 00 00 00 22 00 00 00 00 00 00 00 00 00 00
B800:0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
B800:0090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
BA00:0000 80 00 00 00 00 00 00 08 00 00 00 00 00 30 66 EE
BA00:0010 00 66 00 00 00 00 00 00 30 00 66 00 00 00 00 30
BA00:0020 00 00 00 00 00 14 00 00 00 00 00 00 00 00 00 00
BA00:0030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
BA00:0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
BA00:0050 80 00 00 00 00 00 00 08 00 00 00 00 00 30 7C FE
BA00:0060 00 7C CC 76 C0 CC CC 0C 30 00 C0 CC FE 66 CC 30
BA00:0070 CC 76 00 00 00 41 00 00 00 00 00 00 00 00 00 00
BA00:0080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
BA00:0090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 6

Dump of selected portions of the color/graphics display buffer in high-resolution graphics mode. Each 80 bytes corresponds to one horizontal scan line of the display. The buffer is interlaced: B800:0000 corresponds to Y=0, BA00:0000 to Y=1, B800:0050 to Y=2, BA00:0050 to Y=3, etc.

(Continued from page 35)

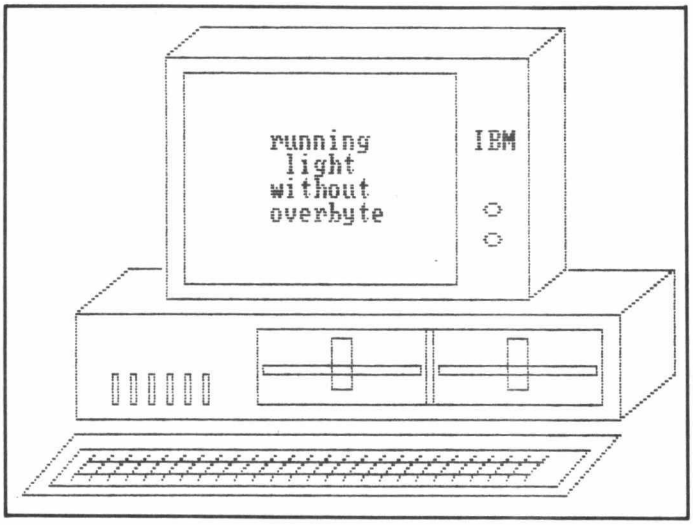


Figure 7

Demonstration of high-resolution graphic image transferred to Epson MX-80 printer with Graftrax option, using the program in Listing 3. [Reduced for publication.]

line-generator subroutines. This algorithm is fairly simple but accurate, and requires as input arguments only the two endpoints of the desired line. It uses the axis with the farthest displacement as the independent variable, stepping through the plot in single units of resolution. Working values of the dependent variable and the slope, along the axis of shorter displacement, are maintained in registers scaled up by a factor of 1000. This ensures that any rounding errors will be less than the resolution of the display.

Listing 1 is in standard Intel 8086 assembly language; those of you who must use the corrupted mnemonics of the Microsoft assembler have my sympathy. To keep the subroutine relatively short I have used the ROM video service routine to plot dots – the user may improve the execution speed significantly by writing a short assembly language sequence to write the bits directly into the memory map. Changing the scaling factor from 1000 to 1024 will allow replacement of the integer multiply and divide instructions with a sequence of arithmetic shifts, which will yield an additional speed advantage.

Listing 2 is the equivalent line-drawing algorithm coded in "high-level" Forth. Note that the subroutine named "!DOT" plots points into the memory map direcly. The two operators "C@L" and C!L" will be unfamiliar to Forth programmers – they are "long" fetch and store operators which require a double-word address consisting of the 16-bit segment or paragraph and the 16-bit offset.

Printing Graphic Displays

The IBM dot matrix printer, which is actually built by Epson, can be enhanced with a special ROM set called "Graftrax" (tm) which gives the device a bit-plot graphics capability. The option costs about $75.00, is available from many computer stores and mail order dealers, and may be installed by the user with a screwdriver in about ten minutes.

I have included as Listing 3 a short program written in Forth which will translate a high-resolution video graphic display into hard copy on any printer equipped with Graftrax. An example of an image printed by this program is shown in Figure 7.

Due to the slight difference in dot spacing, the hard copy image is distorted by a magnification in the vertical axis of about ten percent. The printer is used in the 480-dot mode, thus the right one-fourth of the video display will be omitted. The program may readily be modified to dump the right and left halves of the image separately, they may then be pasted up for further reproduction.

Future Topics

If enough readers express interest, I will be happy to contribute further columns on video graphics, using the IBM PC as th example machine. Possible topics include assembly language and high-level language routines for translation and rotation of figures, use of the color modes, and subroutines for generation of more complex shapes such as cubes, ellipses, and parabolas.

(Listing begins on page 37)

Listing 1

```
                        title   'Hi-Res Graphics Line Draw'
                        pagewidth       80
                        pagesize        66
                        cseg


                ;
                ;
                ; Line drawing routine for
                ; high resolution graphics
                ; display on IBM Personal Computer.
                ;
                ; Ray Duncan   April 1, 1982
                ; Laboratory Microsystems
                ;
                ;
                line:                           ;calling arguments:
                                                ;dx=X1, cx=Y1
                                                ;bx=X2, ax=Y2
                                                ;
0000 8B F0                      mov     si,ax   ;si = copy of Y2
0002 8B FB                      mov     di,bx   ;di = copy of X2
0004 2B C1                      sub     ax,cx   ;ax = Y2-Y1
0006 0B C0                      or      ax,ax   ;if delta Y=0
0008 75 03                      jnz     line04  ; go to
000A E9 AC 00                   jmp     line6   ; fast draw routine
000D 2B DA              line04: sub     bx,dx   ;bx = X2-X1
000F 0B DB                      or      bx,bx   ;if delta X=0
0011 75 03                      jnz     line05  ; go to
0013 E9 BE 00                   jmp     line7   ; fast draw routine
0016 50                 line05: push    ax      ;save actual delta X
0017 53                         push    bx      ; and delta Y
0018 79 02                      jns     line06  ;if delta x negative
001A F7 DB                      neg     bx      ; make it positive
001C 0B C0              line06: or      ax,ax   ;if delta y negative
001E 79 02                      jns     line07  ; make it positive
0020 F7 D8                      neg     ax
0022 3B C3              line07: cmp     ax,bx   ;abs(Y2-Y1) > abs(X2-X1)?
0024 5B                         pop     bx      ;restore original
0025 58                         pop     ax      ; delta y and delta x
0026 7F 48                      jg      line3   ;jump if delta y larger
0028 3B D7                      cmp     dx,di   ;if X2 < X1 then
002A 7C 04                      jl      line08  ; interchange X1,Y1
002C 87 D7                      xchg    dx,di   ; with X2,Y2
002E 87 CE                      xchg    cx,si
0030 52                 line08: push    dx      ;save X1
0031 53                         push    bx      ;save delta X
0032 BB E8 03                   mov     bx,1000 ;scale up delta Y
0035 F7 EB                      imul    bx      ; by 1000
0037 5B                         pop     bx      ;then divide by delta X
0038 F7 FB                      idiv    bx      ; to get Y increment*1000
003A 5E                         pop     si      ;per unit of X
003B 50                         push    ax
003C B8 E8 03                   mov     ax,1000 ;also get Y1*1000
003F F7 E9                      imul    cx
0041 59                         pop     cx
```

(Continued on next page)

```
0042 50                 line1:  push    ax          ;save registers
0043 51                         push    cx          ;cx    =  Y incr * 1000
0044 52                         push    dx          ;dx:ax=  Y * 1000
0045 56                         push    si          ;si    =  X1
0046 57                         push    di          ;di    =  X2
0047 BB E8 03                   mov     bx,1000     ;scale Y back down
004A F7 FB                      idiv    bx          ; to graphable range
004C 8B D0                      mov     dx,ax
004E 8B CE                      mov     cx,si       ;now dx=Y cx=X
0050 B8 01 0C                   mov     ax,0c01h    ;set function 12 code
0053 CD 10                      int     10h         ;call ROM to plot point
0055 5F                         pop     di          ;restore registers
0056 5E                         pop     si
0057 5A                         pop     dx
0058 59                         pop     cx
0059 58                         pop     ax
005A 3B F7                      cmp     si,di       ;whole range of X plotted?
005C 74 11                      jz      line2       ;yes,exit
005E 46                         inc     si          ;increment X
005F 33 DB                      xor     bx,bx       ;sign extend Y increment
0061 0B C9                      or      cx,cx
0063 79 03                      jns     line15
0065 BB FF FF                   mov     bx,-1
0068 03 C1              line15: add     ax,cx       ;and calculate new Y value
006A 13 D3                      adc     dx,bx
006C E9 D3 FF                   jmp     line1       ;go plot next point
                                                    ;
006F C3                 line2:  ret                 ;common exit point
                                                    ;
                                                    ;come here if delta Y
                                                    ; greater than delta X
                                                    ;
0070 3B F1              line3:  cmp     si,cx       ;make sure Y1 < Y2
0072 7F 04                      jg      line4       ;if not, interchange
0074 87 F1                      xchg    si,cx       ; X1,Y1 with X2,Y2
0076 87 FA                      xchg    di,dx
0078 52                 line4:  push    dx          ;calculate X increment
0079 50                         push    ax
007A B8 E8 03                   mov     ax,1000     ;scale up delta X
007D F7 EB                      imul    bx
007F 5B                         pop     bx          ;divide delta X * 1000
0080 F7 FB                      idiv    bx          ; by delta Y
0082 8B F9                      mov     di,cx       ; to get X increment
0084 8B C8                      mov     cx,ax       ; * 1000 per unit Y
0086 58                         pop     ax
0087 BB E8 03                   mov     bx,1000     ;also calc X1 * 1000
008A F7 EB                      imul    bx
008C 50                 line5:  push    ax          ;save registers:
008D 51                         push    cx          ;cx    = X incr * 1000
008E 52                         push    dx          ;ax:dx= X * 1000
008F 56                         push    si          ;si    = Y2
0090 57                         push    di          ;di    = Y1
0091 BB E8 03                   mov     bx,1000     ;scale down current X
0094 F7 FB                      idiv    bx          ; value to graphable value
0096 8B C8                      mov     cx,ax
```

```
0098 8B D7                      mov     dx,di    ;now dx=Y cx=X
009A B8 01 0C                   mov     ax,0c01h;set function 12
009D CD 10                      int     10h      ;now plot point
009F 5F                         pop     di       ;restore registers
00A0 5E                         pop     si
00A1 5A                         pop     dx
00A2 59                         pop     cx
00A3 58                         pop     ax
00A4 3B F7                      cmp     si,di    ;whole range of Y done?
00A6 74 C7                      jz      line2    ;yes,exit
00A8 47                         inc     di       ;increment Y
00A9 33 DB                      xor     bx,bx    ;sign extend X increment
00AB 0B C9                      or      cx,cx
00AD 79 03                      jns     line55
00AF BB FF FF                   mov     bx,-1
00B2 03 C1              line55: add     ax,cx    ;now calculate new X value
00B4 13 D3                      adc     dx,bx
00B6 E9 D3 FF                   jmp     line5
                                                 ;
                                                 ;fast plot routine for
                                                 ;Y2=Y1 i.e. horizontal line
00B9 87 D1              line6:  xchg    dx,cx
00BB 3B D9                      cmp     bx,cx
00BD 7D 02                      jge     line65   ;force X1 < X2
00BF 87 CB                      xchg    cx,bx
                        line65:                  ;save registers:
00C1 53                         push    bx       ;bx = X2
00C2 51                         push    cx       ;cx = X1
00C3 52                         push    dx       ;dx = Y
00C4 B8 01 0C                   mov     ax,0c01h;set function 12 code
00C7 CD 10                      int     10h      ;plot point
00C9 5A                         pop     dx       ;restore registers
00CA 59                         pop     cx
00CB 5B                         pop     bx
00CC 41                         inc     cx       ;increment X
00CD 3B D9                      cmp     bx,cx    ;whole range of X done?
00CF 7D F0                      jge     line65   ;no
00D1 E9 9B FF                   jmp     line2    ;yes,exit
                                                 ;
                                                 ;fast draw routine for
                                                 ;X2=X1 i.e. vertical line
00D4 87 CA              line7:  xchg    cx,dx
00D6 8B DE                      mov     bx,si
00D8 3B DA                      cmp     bx,dx
00DA 7D 02                      jge     line75   ;force Y1 < Y2
00DC 87 DA                      xchg    bx,dx
                        line75:                  ;save registers
00DE 53                         push    bx       ;bx = Y2
00DF 51                         push    cx       ;cx = X
00E0 52                         push    dx       ;dx = Y1
00E1 B8 01 0C                   mov     ax,0c01h;set function 12 code
00E4 CD 10                      int     10h      ;plot point
00E6 5A                         pop     dx       ;restore registers
00E7 59                         pop     cx
00E8 5B                         pop     bx
00E9 42                         inc     dx       ;increment Y
00EA 3B DA                      cmp     bx,dx    ;whole range of Y done?
```

(Continued on page 42)

```
  00EC 7D F0                        jge     line75  ;not yet
  00EE E9 7E FF                     jmp     line2   ;yes,exit

                                    end
END OF ASSEMBLY. NUMBER OF ERRORS:   0
```

Listing 2

```
Screen # 39
  0 ( Plot dot x,y                                              04/23/82 )
  1 FORTH DEFINITIONS HEX
  2
  3 B800 CONSTANT MAP                       ( addr of memory map/16 )
  4 0 VARIABLE BIT-TABLE -2 ALLOT           ( bit masks for dot )
  5 80 C, 40 C, 20 C, 10 C, 8 C, 4 C, 2 C, 1 C,
  6
  7 ( Plot dot at coordinates x,y.  Stack effect:  x y --- )
  8 : !DOT  OVER 7 AND BIT-TABLE +          ( extract bit mask )
  9         C@ >R                           ( and save it )
 10         DUP 1 AND                       ( calc byte addr, if y odd)
 11         IF 2000 ELSE 0 ENDIF            ( then offset by 2000h )
 12         SWAP 2/ 50 * + SWAP 8 / +       ( each dot row=80 bytes )
 13         MAP SWAP 2DUP C@L               ( extract byte from memory )
 14         R> OR ROT ROT C!L ;             ( map, turn on bit and )
 15 DECIMAL -->                             ( put byte back in map )
Screen # 40
  0 ( Draw line from x1,y1 to x2,y2                             04/08/82 )
  1
  2 0 VARIABLE INCR 2 ALLOT                     ( save x or y increment)
  3 : 4DUP 4 PICK 4 PICK 4 PICK 4 PICK ;    ( 4 wd analogue of DUP)
  4
  5 ( Draw line.  Stack effect:  x1 y1 x2 y2 --- )
  6 : LINE  4DUP ROT - ABS ROT ROT SWAP - ABS <    ( y2-y1  x2-x1 )
  7         IF    ( delta x > delta y, so plot x1 to x2 )
  8             4 PICK 3 PICK > IF 2SWAP THEN   ( force x1 < x2 )
  9             4DUP ROT - ROT ROT SWAP -       ( y2-y1  x2-x1 )
 10             SWAP 1000 M* ROT M/             ( save incr*1000)
 11             S->D INCR 2! DROP
 12             DROP SWAP 1000 M*     ( discard y2, take y1*1000 )
 13             4 ROLL 4 ROLL SWAP    ( bring up x1 and x2 )
 14 -->
 15
Screen # 41
  0 ( Draw line from x1,y1 to x2,y2 continued                   04/08/82 )
  1
  2             DO 2DUP 1000 M/ SWAP DROP I      ( now plot points)
  3              SWAP !DOT INCR 2@ D+ LOOP
  4         ELSE (  delta y > delta x, so plot y1 to y2 )
  5              3 PICK 2 PICK > IF 2SWAP THEN   ( force y1 < y2 )
  6              4DUP ROT - ROT ROT SWAP -       ( y2-y1  x2-x1 )
  7              1000 M* ROT M/                  ( save incr*1000 )
  8              S->D INCR 2! DROP
  9              SWAP DROP ROT 1000 M* ( discard x2, take x1*1000 )
 10              4 ROLL 4 ROLL SWAP     ( bring up y1 and y2 )
 11              ( now plot points )
```

```
 12              DO 2DUP 1000 M/ SWAP DROP I !DOT
 13                 INCR 2@ D+ LOOP
 14          THEN  2DROP ;
 15
```

Listing 3

```
Screen # 48
  0 ( Dump hi-res graphic image to Epson printer          04/23/82 )
  1 FORTH DEFINITIONS HEX
  2 B800 CONSTANT MAP                    ( addr of memory map/16 )
  3
  4 0 VARIABLE BIT-TABLE -2 ALLOT        ( bit masks for dot testing)
  5 80 C, 40 C, 20 C, 10 C, 8 C, 4 C, 2 C, 1 C,
  6 0 VARIABLE BIT-BUFFER 1DE ALLOT      ( build printer output here)
  7
  8 ( Fetch dot value from memory map.  Stack effect:   x y --- )
  9 : @DOT  DUP 1 AND                    ( calc byte addr, if y odd)
 10        IF 2000 ELSE 0 ENDIF         ( then offset by 2000h )
 11        SWAP 2/ 50 * + OVER 8 / +    ( each dot row=80 bytes )
 12        MAP SWAP C@L SWAP 7 AND      ( get mask from table to )
 13        BIT-TABLE + C@ AND           ( test dot position )
 14        IF 1 ELSE 0 THEN ;           ( return 1 if dot on,else 0)
 15 -->
Screen # 49
  0 ( Dump hi-res graphic image to Epson printer          04/23/82 )
  1
  2 ( after a pattern has been built into BIT-BUFFER )
  3 ( corresponding to x=0 to 479 & y=y to y+3  by )
  4 ( DUMP-MAP, this routine sends it to the printer )
  5
  6 : SEND-LINE      0A EMIT             ( line feed )
  7                  1B EMIT 4B EMIT     ( select graphics mode )
  8                  E0 EMIT 01 EMIT     ( 480 bytes will be sent )
  9                  1E0 0 DO
 10                    BIT-BUFFER I + C@ EMIT  ( send this byte )
 11                 LOOP ;              ( loop through bit buffer )
 12 -->
 13
 14
 15
Screen # 50
  0 ( Dump hi-res graphic image to Epson printer          04/23/82 )
  1
  2 : DUMP-MAP       C8 0 DO             ( for y=0 to 199 step 4 )
  3                    1E0 0 DO 0        ( for x=0 to 479 )
  4                      I J @DOT IF 80 OR THEN       ( get x,y )
  5                      I J 1+ @DOT IF 20 OR THEN    ( get x,y+1 )
  6                      I J 2+ @DOT IF 08 OR THEN    ( get x,y+2 )
  7                      I J 3 + @DOT IF 02 OR THEN   ( get x,y+3 )
  8                      BIT-BUFFER I + C!            ( save pattern)
  9                    LOOP      ( next x )     SEND-LINE
 10                  4 +LOOP ;   ( next y )
 11
 12 : DUMP           PRINTER                    ( select parallel port)
 13                  1B EMIT 41 EMIT 08 EMIT    ( set vertical spacing)
 14                  DUMP-MAP CONSOLE ;         ( send graphics image)
 15 DECIMAL ;S
```

End Listing

More on Checksums

As a follow-up to our recent article on checksums, DDJ *April 1982, here is a discussion of some additional examples.*

A personal check is probably the most common example of a document that contains information which must be accurately reproduced several times through mechanical or electronic means on its journey from, for example, the receiving bank through the Federal Reserve System, and finally to the bank on which it is drawn. At each of these processing steps the accuracy of the routing information must be verified or the chances of it being falsely routed are increased.

Certainly the most reasonable method of verification in this example is the check digit. Examine a blank check for a moment. Along the bottom are a series of pre-encoded fields. The third from the right is normally the routing and transit number. In the U.S. it will always be in one of two formats. Either nine digits long or two groups of four digits separated by a dash. In the latter case, no check digit is built in and the only absolute verification method available would be comparison to all known number series.

Referring to Figure 1, the check digit calculation example can be used to verify the routing number on any check drawn on a U.S. bank. Try it on yours!

Another common example of information verification through check digits is oil company credit cards. The oil industry is faced with problems similar to the banking industry, rapid verification of information produced through mechanical or electronic means. If you have an oil company credit card with an account number grouped in a 3-3-3-1 pattern, refer to Figure 2 for an example of a check digit calculation which will probably work with your card.

Figure 3 will be helpful in categorizing the type of error each check digit type is most proficient at discerning.

Finally, check digits are the most common form of verification of decimal numbers in existence today. Proceeding one step further, it is possible to reconstruct any single digit of a number string providing the check digit is present and all other digits of the original number are correct. Considering the obvious advantage of human interface inherent in the decimal numbering system, one of the best encoding systems is bar-coded binary data with error correction built in which can easily correct many missing digits. Five or six digits of correction is not uncommon. ■■J

Figure 1

Modulus:	10
Weights:	3,7,1,3,7,1,3,7
Bank number including a check digit in the 9th position:	061000010

Example

	0	6	1	0	0	0	0	1	
	3	7	1	3	7	1	3	7	
	0	42	1	0	0	0	0	7	Sum = 50
Add to units position to produce zero results									0

0 = Check Digit

Figure 2

Basic account number	4	0	7	5	5	0	5	6	1	2
Multiply units position and every alternate position by two	2	1	2	1	2	1	2	1	2	
	8	0	14	5	10	0	10	6	2	
Cross add all digits	8+0+1+4+			5+1+0+0+			1+0+6+2			=28
Subtract units position of result from 10	10−8=2									

Check Digit is 2

by Robert Blum

Robert Blum, 5536 Colbert Trail, Norcross, GA 30092.

Figure 3

Type of Check Digit Numbering System	Type of Error Detected	Type of Error Not Detected
Modulus 11	100% of all transcription errors.	Approximately 10% of random errors.
Geometric Weights	90% of random errors, 100% single and double transposition errors.	
Arithmetic Weights	All single and double transposition errors, 90% of random errors.	Certain random errors from erroneous recording.
Modulus 10	All transcription errors	Certain types of random errors.
1-3-7	88% of transposition errors. 88% of double transposition errors. 90% of random errors. High percentage of alternate column transpositions.	
Double ADD Double	100% transcription error. 97% transposition error. 90% random error.	Transpositions in adjacent positions between 09 and 90. Alternate column transposition of interchanged digits. Random error caused by substitution of a valid check digit with an erroneous basic account number.

Part II

PDP-1802

Part I of this article (*DDJ* #68) described the instruction set of the PDP-8 minicomputer, and presented a program which emulates the front panel functions of the PDP-8 on microcomputers using the RCA COSMAC 1802 microprocessor. In this concluding part, the emulator's instruction interpreter is described, together with the input/output facility.

Input/Output on the PDP-8

I/O instructions on the PDP-8 have the op code 6. DEC has designated a large number of codes for the remaining bits of the I/O instructions corresponding to the selection codes and operating requirements of peripheral devices of all kinds, including terminals, disk drives, tape drives, oscilloscope displays, paper tape reader/punches, and so on. The most elementary I/O instruction group is concerned with the Teletype unit used as a console on PDP-8s, and these are the only instructions explicitly realized in the emulation.

Like the Operate instructions described in Part I, the I/O instructions can be microprogrammed to combine multiple actions in single instructions. The Teletype I/O instructions allow for a total of three actions for input and three actions for output which may be combined.

The PDP-8 used as its console a modified Teletype Model 33 ASR. This Teletype unit has a keyboard, a printer, a paper tape reader, and a paper tape punch. The printer and the punch are, so far as the computer is concerned, one device; the punch is turned on and off manually, and identical data are acted on by the printer and punch. The paper tape reader and the keyboard are similarly seen by the computer as one source of data. There are, however, important differences in operation which the emulator must take into account if standard PDP-8 software is to be executed correctly. First, the keyboard is set up to make the eighth most significant bit of the ASCII code always marking (that is, always a 1). The paper tape reader, however, does not do this, and must correctly sense and transmit whatever actually is present on the tape in the eighth position. If you plan to use a Teletype in conjunction with this emulator, you must be sure that it works this way, or modify it accordingly.

Second, the paper tape reader on the Teletypes supplied by DEC for use with PDP-8s are equipped with a relay which is opened or closed as a function of a flag in the computer. The relay permits or inhibits advance of the paper tape reader; since the flag controlling the relay may be sensed or cleared under program control, this allows the computer to read some data, then stop the tape reader while taking an action, then read more data when ready. This capability is used, for example, when assembling a program: source code is read a line at a time from the reader, which is then stopped while the line is assembled and the output printed or punched or both.

In the emulator, a similar signal must be supplied to provide control over the paper tape reader. Although it is common to use the Q line for such purposes in 1802 systems, a more general solution is preferable, and in any case the Q line is used for displaying the link bit. Accordingly, one bit in an output port is chosen as the relay control flag.

Keyboard Instructions

The general form of the keyboard/reader instructions is $603X_8$ where the least significant three bits represented by "X" control the instruction's three actions.

KSF-6031: Skip on keyboard flag. The next instruction is skipped if the keyboard flag is raised, indicating that an ASCII character has been loaded into the keyboard buffer (the buffer content could be binary data too, of course).

KCC-6032: Clear the accumulator, and clear the keyboard flag.

KRS-6034: Move the content of the keyboard buffer into the accumulator.

The usual microprogrammed combination of these is:

KRB-6036: Clear the keyboard flag, and move the content of the keyboard buffer into the accumulator.

A common use of these instructions is to set up a loop (using KSF) which the program exits when a character has been assembled in the buffer (raising the flag); the program then obtains the character for processing, and clears the flag (with a KRB).

Printer/punch Instructions

The three printer/punch microinstructions resemble those for the keyboard and reader; their general form is $604X_8$.

TSF-6041: Skip on printer flag. The next instruction is skipped if the printer flag is raised. The flag is raised upon completion of printing or punching a character from the printer buffer.

TCF-6042: Clear the printer flag.

TPC-6044: Move the content of the accumulator to the printer buffer, then transmit the character to the printer/punch. The printer flag is raised when the action has been completed.

Again, these actions may be combined, as follows:

TLS-6046: Clear the printer flag; transfer the content of the accumulator to the printer buffer; transmit the character to the printer, and raise the printer flag upon completion of the action.

As with the keyboard/reader instructions, a common use of the printer instructions is to create loops in which the program remains until the printer has completed processing of a character. This can be used to ensure that the printer is given something to do only when it is ready.

It is important to note that the activities of the keyboard/reader and the printer/punch are wholly independent of one another. With appropriate programming, one character may be received for processing while another is being printed or punched.

Interrupt Processing

The approach to servicing Teletype input and output described above depends upon having the computer sit in loops while it waits for the very slow

(Continued on page 50)

by Chris Hamlin

Chris Hamlin, 2202 Hindle Ln., Bowie, MD 20716.

(Continued from page 46)

devices to complete actions in progress. To avoid such wastes of time (during which the computer could be doing useful work) an interrupt facility is used. In the PDP-8 peripheral devices may raise interrupt requests at any time; programming must then determine the source of the request and, if necessary, resolve competition between interrupting devices for system resources. In the PDP-1802 emulator, the interrupt action is provided for only in the case of the keyboard and printer I/O devices in the Teletype (and thus the reader and punch, too). First, the interrupt action in the PDP-8 is described. Then ways of emulating it are discussed and presented.

Interrupts are issued whenever the keyboard flag or the printer flag are raised. When the interrupt has been requested, the computer finishes executing the current instruction. Then, instead of proceeding to the instruction it would normally have executed next, it automatically executes a "JMS 0000" instruction. Remember that this will place the address of the next instruction in the interrupted program in location 0000, which is the normal action of a JMS instruction. After the JMS 0000 has been performed, the instruction in location 0001 will be executed, and this is, as a rule, a jump to the interrupt service routine you have provided. To return to the interrupted program after the completion of the service routine, you need merely issue a "JMP I 0000" (indirect jump via location 0000) instruction.

When an interrupt occurs, all further interrupts are disabled until the program (generally the interrupt service routine) enables them again. This ensures that an orderly progression in service for time-critical applications may be guaranteed. Two instructions are provided to control the interrupt facility:

ION-6001: The interrupt facility is enabled, but only following execution of the next instruction (after the ION); this allows exit from the interrupt service routine to occur before another interrupt is received for processing.

IOF-6002: The interrupt facility is disabled. Interrupt requests are ignored.

Note that the occurrence of an interrupt automatically disables the facility; the program need not issue an IOF.

The interrupt facility of the PDP-8 could be emulated in either of two ways. One approach would be to use the actual interrupt apparatus on the 1802; then the interrupt service routine handling 1802 interrupts could be set up so as to carry out exactly the same actions taken by a PDP-8. Another approach would be to simulate interrupts using the input flags (the so-called EF lines) on the 1802. The emulator software could then simply check the state of the flags after executing each PDP-8 instruction, and take appropriate action.

The simulated interrupts have the disadvantage of requiring additional code and therefore additional overhead on every emulated PDP-8 instruction cycle. For many 1802 users, however, this is preferable, because in order to use the true interrupts of the 1802 to emulate the PDP-8 facility some additional hardware is required. To make the emulator usable on unmodified 1802 systems, therefore, the simulated interrupt approach is adopted here for the printer and the keyboard/reader.

Serial Communications Facility

Standard PDP-8 computers were supplied with a hardware facility for converting eight-bit characters to a Teletype-compatible serial stream, and for receiving serial Teletype characters and converting them to eight parallel bits. Today the devices for carrying out this parallel-to-serial and serial-to-parallel conversion are called UARTs (for Universal Asynchronous Receiver Transmitter). In the PDP-1802, the action of the PDP-8 serial communications facility is emulated by a software UART (that is, a program providing the necessary conversion and timing). This is provided so that users without their own UARTs (hardware, that is) can still use the emulator.

A major shortcoming of the software UART approach should be pointed out, however. The whole point of the PDP-8 interrupt architecture and of the serial communications facility was to make the very slow Teletype operations able to proceed while the CPU did a great deal of useful work on other matters; the attention of the CPU is thus only required at the beginning or end of a slow I/O operation, but not during it. The software UART implemented in the 1802 emulation, on the other hand, ties up the 1802 CPU *during* character reception and transmission, with the result that no other business can be taking place at the same time. Many readers may find that it is preferable, for this reason, to go to the slight additional expense of adding a hardware UART (such as RCA's CMOS 1854, which is made for use with the 1802) so that true interrupt benefits are realized.

Other PDP-8 Capabilities

The PDP-1802 does not emulate all of the PDP-8 capabilities. The most important one which is not emulated is the DMA facility. DMA (direct memory access) is a means whereby extremely rapid, controlled data transfer can be provided without having to execute instructions in the CPU. The PDP-8 was unusually powerful for its time and cost, in large part because of the flexible DMA facilities it provided. Disk drives and real-time laboratory instrumentation can be attached quite easily to the DMA facility, and this made the PDP-8 attractive for many applications previously requiring larger, more costly computers. Although the 1802 also has a data channel which closely resembles DMA in its operation, the hardware complexity required to emulate 12-bit DMA on an 8-bit device is excessive, and is unnecessary for executing the basic PDP-8 instruction set on the emulator.

Later models of the PDP-8 incorporated a number of enhancements, and also introduced subtle changes in the actions occurring within the computer when certain instructions were executed. The most important extension to the instruction set, which is not included in the PDP-1802 (although it could easily be added) was the PDP-8/E instruction BSW (7002_8), which swaps the upper and lower halves of the accumulator (a byte swap). The PDP-8/E also included additional interrupt control, teletype/punch and keyboard/reader instructions. If you obtain PDP-8 software to run on the emulator, be sure to ascertain whether it contains any of these unsupported instructions, and provide patches to replace them.

Running the Emulator

When a program is loaded, type "L" and the starting address; then press "S", and program execution commences. You

may at any time begin to single step through it as described in Part I (*DDJ* #68). When you single step, the program counter, memory buffer, accumulator, and instruction buffer are displayed as each instruction is executed. A patch is given allowing display of these registers on the console device as normal execution proceeds.

The Q line is used to show the state of the link bit at the conclusion of each instruction; the low eight bits of the accumulator are also shown in hex on two seven-segment LEDs (or eight single LEDs if you use them). If you do not have a CRT to devote to use as the console for the monitor, you may use your hex keypad or toggle-switch panel, plus the LED displays, to supply the octal console monitor functions instead.

Notes on the Implementation

The octal console monitor is made up almost entirely of subroutine calls using the RCA standard call-and-return technique. The emulator for the PDP-8 instructions, on the other hand, is almost entirely in-line code, the only subroutines being those for moving between the eight-bit and twelve-bit address spaces. The 1802 architecture, whose most important feature is a complement of sixteen registers of sixteen bits each, is very well suited to emulating the twelve-bit architecture, since most of the internal registers of the PDP-8 can simply be represented directly by 1802 registers. Many points of the emulation are greatly simplified, too, by the ability of 1802 registers to change function (pointer, counter, data storage, and so on) as needed.

Needless to say, this is not a high-speed approach to computation. It seems fair to say that for most applications, the emulator will operate between several hundred and several thousand times slower than would the slowest PDP-8 (the serial architecture PDP-8/S). However, it can be used to gain real insight into the operation of the PDP-8 instructions, and will provide faithful execution of correct PDP-8 software composed of the supported instructions. The most complex piece of software I have run using the emulator has been a floating-point package adapted from one distributed by DECUS (the DEC Users' Society) which processes five-word numbers with 51-bit mantissas. This package works flawlessly, albeit a trifle sluggishly.

I would be most interested to hear from readers who implement the PDP-1802, and will answer any questions or try to help with problems others may encounter as best I can.

(Listing begins on page 52)

(Note: The entry point is 1400H. The program assumes the program counter is in Register 0.)

I/O and Console Routines

```
11 90 12 F8 18 F2 32 BF F8 01 F2 32 A1 8F FA 80 32 A1
11 A0 18 F8 02 F2 32 B1 F8 00 AC 8F FA 7F AF 9C FA 10
11 B0 BC F8 04 F2 32 BC 8D AC 9C FA 10 BC C0 14 23 F8
11 C0 20 F2 32 BC F8 01 F2 32 CF 8F FA 08 32 CF 18 F8
11 D0 02 F2 32 D8 8F FA F7 AF F8 04 F2 32 E6 8C AD D4
11 E0 12 B2 8F F9 0C AF C0 14 23 00 00 00 00 00 00 00
11 F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Monitor Driver

```
12 00 3E 00 6A FA 7F AD D5 46 AD 8D 52 F8 04 B0 20 90
12 10 3A 0E 67 22 D5 FC 00 F8 80 AD 36 1A F8 01 B0 F8
12 20 A5 A0 20 90 3A 22 F8 02 B0 F8 47 A0 20 90 3A 2C
12 30 8D 3E 35 FF 00 76 AD 3B 26 F8 03 B0 F8 41 A0 20
12 40 90 3A 3F D5 F8 80 BD 7B F8 02 B0 F8 47 A0 20 90
12 50 3A 4E 8D 76 AD CF 7B 38 7A 9D F6 BD 3B 48 7A F8
12 60 04 B0 F8 42 A0 20 90 3A 65 D5 D3 E2 96 73 86 73
12 70 93 B6 83 A6 46 B3 46 A3 30 6A D3 96 B3 86 A3 E2
12 80 12 72 A6 F0 B6 30 7A 00 00 00 00 00 00 00 00 00
12 90 3E 9A 8F F9 84 AF 6A F9 80 AD C0 15 00 00 00 00
12 A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
12 B0 46 AD 8D 52 67 22 F8 04 B0 20 90 3A B9 D5 00 00
12 C0 D3 E2 96 73 86 73 93 B6 83 A6 46 B3 46 A3 30 C0
12 D0 D3 96 B3 86 A3 E2 12 72 A6 F0 B6 30 D0 00 00 00
12 E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
12 F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E2 22
13 00 D4 13 BA D4 12 00 D4 13 C3 43 CB 13 03 D4 13 59
13 10 9B B9 8B A9 30 1D 99 BB 89 AB D4 13 90 D4 16 00
13 20 D4 16 1A D4 12 07 20 D4 13 90 D4 12 07 20 D4 12
13 30 00 8D FB 20 32 53 D4 13 C3 4C 3B 0D D4 13 C3 53
13 40 CB 15 ED D4 13 C3 43 CB 13 03 D4 13 59 D4 16 00
13 50 D4 16 35 19 D4 13 BA 30 16 F8 04 BD 8D FF 30 3B
13 60 72 FF 08 33 72 8D 52 67 22 FA 07 73 9D FF 01 32
13 70 77 BD D4 12 00 30 5C 12 72 AB 02 FE FE FE 52 8B
13 80 F1 AB 12 02 76 76 76 52 8B F1 AB 12 02 7E BB D5
13 90 9B F6 F9 30 AD D4 12 09 8B 7E 7E 7E FA 07 F9 30
13 A0 AD D4 12 09 8B F6 F6 F6 FA 07 F9 30 AD D4 12 09
13 B0 8B FA 07 F9 30 AD D4 12 09 D5 D4 12 07 0D D4 12
13 C0 07 0A D5 46 52 8D F3 3A D1 F6 D4 12 09 D4 13 BA
13 D0 C8 FF 00 D5 AD D4 12 B2 D5 00 00 00 46 52 8D F3
13 E0 32 E6 FF 00 30 EC F6 67 22 D4 13 F7 D5 00 00 00
13 F0 3E F0 6A FA 7F AD D5 D4 12 B0 0D D4 12 B0 0A D5
```

Initialization
Single Step

```
14 00 90 B3 F8 06 A3 D3 F8 12 B4 B5 F8 6B A4 F8 7B A5
14 10 E2 F8 18 B2 F8 FF A2 F8 14 B1 F8 00 A1 C0 13 00
14 20 C4 C4 12 9F FA 80 C6 3F 6E 37 29 3F 2B 6C 64 22
14 30 BE FA 80 BF C4 C4 C4 37 37 6C 64 22 AE 8C 52 64
14 40 22 88 AB 98 BB D4 13 90 D4 12 07 20 89 AB 99 BB
14 50 D4 13 90 D4 12 07 20 8C AB 9C FA 0F BB D4 13 90
14 60 D4 12 07 20 8A AB 9A BB D4 13 90 D4 13 BA 8C 52
14 70 64 22 9C FA 10 CE 7B 38 7A C0 12 90 99 B8 30 23
```

```
14 80 F8 01 F2 CA 14 F4 F8 78 F2 32 D1 22 52 FB 08 32
14 90 CB 9C FA 08 32 9C 8F FB 40 AF 30 A8 8C 3A A8 9C
14 A0 FA 0F 3A A8 8F FB 20 AF 9C FA 10 32 B1 8F FB 10
14 B0 AF 02 FA 08 3A BE 8F FA 70 F2 32 D0 30 CB 8A FA
14 C0 70 52 8F FA 70 FB 70 F2 F3 3A D0 18 98 FA 0F B8
14 D0 12 F8 80 F2 32 DD 9C FA 10 BC F8 00 AC F8 04 F2
14 E0 32 F1 8E 22 73 9E FA 0F 52 9C F1 12 BC 8C F1 12
14 F0 AC 8F FA 8F AF F8 02 F2 C2 14 23 7B 00 00 22 AA
```

Interpreter

```
15 00 98 B9 88 A9 18 98 FA 0F B8 D4 16 00 D4 16 1A 8B
15 10 AA 9B BA FA 0C FB 0C C2 16 5A 8A FA 80 3A 33 F8
15 20 00 B9 8A A9 FA F0 3A 3C 89 FA 08 32 3C 8F F9 02
15 30 AF 30 3C 8A FA 7F 52 89 FA 80 F1 A9 9A FA 01 32
15 40 5F D4 16 00 D4 16 1A 8F FA 02 32 5B 1B 9B FA 0F
15 50 BB 9B B9 8B A9 27 D4 16 35 30 5F 9B B9 8B A9 8F
15 60 FA FD AF 9A F6 FC 74 A1 93 B1 01 A3 F6 FC 74 A1
15 70 93 B1 01 A3 7A A7 90 BD D1 E6 D4 16 00 D4 16 1A
15 80 8B 73 9B F9 10 52 9C F2 12 BC 8C F2 AC C0 14 23
15 90 D4 16 00 D4 16 1A 1B 27 8B 3A A1 9B FA 0F 3A A1
15 A0 18 D4 16 35 C0 14 23 D4 16 00 D4 16 1A 9B 73 8B
15 B0 52 8C F4 12 AC 9C 74 FA 1F BC C0 14 23 D4 16 00
15 C0 9C BB 8C AB D4 16 35 9C FA 10 BC F8 00 AC C0 14
15 D0 23 D4 16 00 98 BB 88 AB D4 16 35 19 99 FA 0F B9
15 E0 B8 89 A8 C0 14 23 99 B8 89 A8 C0 14 23 F8 00 AC
15 F0 BC AF BF 89 A8 99 B8 C0 14 23 82 92 21 22 22 92
16 00 22 FC 00 89 FE 73 99 FA 0F 7E 73 12 12 89 F4 73
16 10 99 74 52 72 F6 B7 72 76 A7 D5 47 BB 07 33 2F FA
16 20 F0 F9 08 AB 9B F6 BB 8B 76 AB 3B 24 FC 00 D5 AB
16 30 9B FA 0F BB D5 9B FA 0F 33 4F F9 10 BB 8B FE AB
16 40 9B 7E BB 3B 3D 57 17 8B 52 07 FA 0F F1 57 D5 52
16 50 07 FA F0 F1 57 17 8B 57 D5 F8 8A 73 9A 52 F8 02
16 60 F2 C2 11 90 F8 01 F2 12 CA 14 80 F8 80 F2 32 77
16 70 9C FA 10 BC F8 00 AC F8 40 F2 32 80 9C FA 0F BC
16 80 F8 20 F2 32 8D 8C FB FF AC 9C FB 0F BC F8 10 F2
16 90 32 96 9C FB 10 BC F8 01 F2 32 A7 22 9C FA 10 52
16 A0 1C 9C FA 0F F3 BC 12 F8 08 F2 32 C6 F8 02 F2 52
16 B0 9C F6 BC 8C 76 AC 33 BA 30 BE 9C F9 10 BC 02 C2
16 C0 14 23 F3 52 30 B0 F8 04 F2 C2 14 23 F8 02 F2 52
16 D0 8C FE AC 9C 7E BC FA 20 32 E2 8C F9 01 AC 9C FA
16 E0 1F BC 02 C2 14 23 F3 52 30 D0 DC C1 10 8F D8 6D
16 F0 5D 5D 3E C1 58 02 2B 72 08 67 B6 37 11 B2 67 60
```

Checksum Code*

```
17 10 F8 06 B5 F8 6F A5 F8 10 A7 F8 11 B2 F8 18 B4 F8
17 20 90 A2 E2 F8 00 A4 B6 A6 A8 88 F4 A8 12 27 87 25
17 30 3A 29 F8 10 A7 88 54 E4 86 F4 14 A6 E2 95 32 44
17 40 F8 00 30 2A 86 E4 54 64 24 00 00 00 00 FF 00 FF
```

Checksum Values**

```
18 00 9F E8 22 8E 40 FE 00 D1 B8 F0 27 C7 79 69 15 7A
18 10 F8 00 7F FB F7 00 04 95 BE 7B 68 C5 CF C9 AE AD
18 20 85 CF 06 8F CE 65 AB 49 66 E2 93 B4 92 8D 2D CB
18 30 92 67 17 86 7F 29 9F 6C A7 CF D4 24 2F 36 FB 71
18 40 15 B5 56 87 26 FD AE BF 47 6F 52 16 E2 C3 43 57
18 50 09 AA FC 81 02 9C 96 DE
```

***Execute this code to generate checksum values.**

****One checksum for every line (16 values) of code from 1190H to 16FFH; value at 1857H is checksum for the whole interpreter.**

End Listing

Rough Approximation

Numerical Algorithms

When a numerical calculation is done by a computer, few users know how the numbers are being stored or what algorithms are being applied. This is part of the underground lore of binary digital computation, itself derived from the even more abstruse techniques of higher mathematics. Many users are aware that the result will often be a rounded-off approximation. Few will know anything of approximation theory, that optimizes the trade-off between algorithmic simplification and loss of precision. Some elements of it are intelligibly discussed in *Numerical Methods*, by G. Dahlquist and A. Bjorck (1974, Prentice-Hall, 573 pp.), computer-oriented Swedish mathematicians. An approximation algorithm can be designed to yield a numerical value with either an absolute or a relative error less than some pre-set value, over a defined *interval* of the numerical input. It is not unusual, when the range of input is very wide, to have several different approximation algorithms, each one valid for a different interval. This note will consider only one simple example, the calculation of the natural logarithm of a real binary number, N. The term "rough" approximation means that the relative error, in the interval over which the algorithm is valid, does not exceed 0.1%. However, a similar approach could be used to design algorithms with any desired degree of precision.

Any real binary number N can be most compactly stored in exponential form, $2^n(1 + m)$, where n is a signed integer and m is a binary fraction; $(1 + m)$ must be positive for calculation of the natural log, ln(N), that is

$$\ln(N) = n \ln(2) + \ln(1 + m)$$

The first term is easily calculated by multiplying a look-up table value of ln(2) by n. The second term can be evaluated to any desired precision by the infinite-series algorithm

$$\ln(1 + m) = m(1 - m/2 + m^2/3 - m^3/4 \ldots)$$

but this converges extremely slowly as the fraction m approaches 1. As successive terms are calculated, the error alternates between positive and negative. E.g., if one wishes to simplify calculation by not going beyond the $m^2/3$ term, the error is always positive and worsens as m becomes larger. This can be mitigated by replacing $m^2/3$ by a term that will be numerically less. One computationally-simple algorithm is

$$\ln(1 + m) \approx m[1 - m/2 + (m/2)^2 + (m/2)^2/16]$$

Since all the divisions are by multiples of two, they can be quickly done as right-shifts of m. The only complex operations are two binary multiplications. The first calculates $(m/2)^2$ and the second multiplies the term in brackets by m.

The performance of this algorithm was tested on a pocket calculator. The relative error is low and negative for values of m less than 0.1. At 0.1 it is -0.05%, at 0.2 it is -0.11%, at 0.25 also -0.11%, then declines to -0.07% at 0.3, to -0.01% at 0.34, changes sign to +0.01% at 0.35, +0.06% at 0.37, and +0.16% at 0.40. This limited working interval (here from 0 to 0.37) is typical for approximation algorithms. It is obvious that replacing the $(m/2)^2/16$ term by a slightly larger term would narrow the working interval and lower the maximum negative error.

A different algorithm must be used for values of m greater than 0.37. Again there are very many possible ones. One approach is to calculate $y = (1 - m)$, so that $(1 + m) = (2 - y)$. An approximation algorithm for $\ln(2 - y)$ is

$$\ln(2) - (y/2)[1 + y/4 + (y/4)^2 + (y/4)^2/2 + (y/4)^2/4]$$

Although more complex, the real work still involves only two binary multiplications. The relative error is low and negative for values of m larger than 0.7. At 0.7 it is -0.04%, at 0.6 it is -0.08%, at 0.5 it is -0.1%, at .45 it is -0.9%, at 0.4 it is -0.04%, then changes sign to +0.07% at 0.35. The working interval is twice as wide as that of the first algorithm, because the approximation is in effect a function of m/2. Both algorithms show a negative-error plateau as the difference from their respective "reference points" (ln(1) or ln(2)) increases, and this shifts to a positive error near the limit of the working interval, that is close to the logarithmic mean of the reference points. Near the reference points, where m or $(1 - m)$ becomes very small, even simpler algorithms will do. E.g.,

$$\ln(1 + m) \approx m(1 - m/2 + m/32)$$

has a working interval from zero to 0.12. The problem becomes the relative complexity of selecting one of many simple operations versus one of two more complex ones. It's unlikely that algorithms as "rough" as these have ever been programmed, but more complex approximations are the hallmark of high-speed computer math. Like most fascinating algorithms, they tend to be deeply buried in a mass of machine code, protected by anti-copying tricks. Competition breeds secrecy. DDJ

by H.T. Gordon

H. T. Gordon, College of Natural Resources, University of California, Berkeley, CA.

OF INTEREST

by Michael Wiesenberg

For Your IBM PC

How about the C language for your IBM Personal Computer? Lifeboat Associates, those wizards who have implemented CP/M on virtually every microcomputer, have announced their **Lattice C Compiler** for the IBM PC with DOS. For those who have just returned after ten years of COBOL programming in Ulan Bator, C was developed at Bell Laboratories concurrently with the UNIX operating system. C is both a high- and low-level language that offers control over the computer typically found only in assembly languages. (And did you know that before their successful C language, Bell Labs had A and B?) Lifeboat's C compiler is a full implementation of Kernighan and Ritchie's specification of C as described in their standard text. Lattice C is compatible with UNIX Version 7 and BD Software C Compiler. It produces relocatable machine code in Intel's 8086 object module format. A compile-time option controls whether this code is generated for the 8087 floating-point processor directly or for library functions that simulate its operation. The Lattice C Compiler includes a library, and I/O subroutines for IBM PC DOS and MS-DOS. **(Reader Service No. 314)**

IBM sells programs for the PC written by outside software companies. **Inventory Control**, by Peachtree software, assists a small business to improve merchandise control, reduce inventory investment, and improve customer service and response. It can be used alone, or with Peachtree's General Ledger, Accounts Receivable, and Accounts Payable, and is easily customized. $595. **(Reader Service No. 399) Time Manager Program**, by Microsoft, is a personal calendar that automatically sorts appointments and displays them chronologically, and a "to-do" list that reminds the user of important commitments, organized by priority, with high-priority items not completed on their day heading the list tomorrow. Time Manager totals and accounts income received, hours worked, and money and time expended, giving account totals for specified time periods. It produces, for example, year-to-date totals for all business expenses charged to credit cards or paid by cash or check. $100. **(Reader Service No. 409)** Microsoft has also written a COBOL compiler for the PC, ANSI 74 level standard, with extension for color and screen formatting. It chains to other programs, can combine object modules with those from the IBM PC Macro Assembler, and offers segmentation to run programs larger than memory. $700. **(Reader Service No. 419) CP/M-86**, by Digital Research, is now available, supporting up to two physical or four logical drives, with disk formatting, file copying to other devices, dynamic memory management, and offering an editor, assembler, debugger (DDT86), and a program that produces command files. CP/M-86 is not compatible with IBM PC DOS or the UCSD p-System, and existing CP/M-86 programs running on other computers must be converted to run on the IBM PC, but, they say, "This is normally performed by the program author as part of the development activity." $240. **(Reader Service No. 429)** IBM offers its own **Dow Jones Reporter**, which accesses the Dow Jones News/Retrieval Service, with headlines and news stories for the Wall Street Journal, Barrons, and the Dow Jones News Wire Service, current day market quotes, historical stock quotes, and company profiles and extracts and financial statistics on thousands of companies and 180 industries. This is a software product; users supply modems and cables. **(Reader Service No. 439)** An updated **VisiCalc** (1.1), by VisiCorp, that accomodates 256K and supports additional parallel and serial printers, will be provided free to warranty registrants. **(Reader Service No. 445)** All programs are available from authorized dealers: IBM Product Centers, ComputerLand Stores, and Sears Business Systems Centers, and in volumes of twenty or more through the IBM National Accounts Division.

Baby Blue CPU Plus is a circuit board and software from Xedex that makes the IBM PC compatible with CP/M-80-based programs and supports serial printers. Xedex has acquired OEM (original equipment manufacture) rights to MicroPro software, including WordStar and MailMerge, configured to recognize the IBM keyboard's special keys. Normally WordStar requires user modification and cannot take advantage of the IBM PC's keyboard. The two software products, normally sold together for $575, are made available with Baby Blue as a $980 package. The Baby by itself is $600. **(Reader Service No. 324)**

The FinalWord is a word processor, written in C, for the IBM PC, from Mark of the Unicorn. It can be customized by users, supports (they claim) any printer, has a table of contents program, indexing, footnoting, numbering by chapter/section, deletes/moves letter, word, sentence, paragraph or page, has multiple buffers and windows, deletion recovery, crash recovery, state save, true proportional spacing, and everything else you'd expect from a good WP. The FinalWord is also available on most other micros, and costs $300. **(Reader Service No. 334)**

PHONE SUPPORT, by Information Unlimited Software, for the IBM PC, puts users of IUS's EasyWriter on line (during normal business hours, Pacific Time, with a toll-free 800 number) with service representatives who also help with PC configuration and simple testing procedures. Single users pay $100 a year, and corporations $175. A newsletter is included in the price. **(Reader Service No 344)**

For Your CP/M DOS

FNB Software offers several CP/M utilities on disk for most computers. **SMARTKEY** adds intelligent keyboard facilities to a terminal by intercepting calls to the BIOS, so that individual keys can be redefined to different character codes from that produced by the hardware. For example, cursor control keys can be redefined to suit the cursor commands of word processors. Keys can be defined to represent strings of any length, giving a terminal without them function key capabilities. The current set of definitions can be saved in a disk file or a new set loaded without removing SMARTKEY from memory. $39. **SMARTPRINT** uses definition tables generated by SMARTKEY to translate characters sent to the system printer, permitting use of non-standard printwheels without manual character translation, use of non-ASCII printers or those that cannot generate the full ASCII character set, and substitution of a character sequence for an unavailable character (for example, ".GT." for ">"). $19. **FAST** speeds up systems with slow disk drives, automatically relocating itself to the top of available memory, where it modifies the linkages to the driver routines. Once installed, it is transparent to the user, except for reduced disk activity and throughput, they say, "comparable to that of a hard disk system." WordStar, for example, runs three times as fast. The only penalty is a reduction in memory of 7.25K. Since FAST loads below CP/M, it can be removed without changing disks or SYSGENing a new system. $49. The **Extension Utility Package** includes

SPOOL, to redirect program output from printer, punch, or console to a disk file or both, **UNSPOOL**, to print as a background activity, **QSUB**, to "quick submit" in one line of input several CP/M commands, and **RESTORE**, to recover files inadvertently ERAsed, perform consistency checks on the directory, and sort a directory by name or type while removing erased files. $39. **UNPROTECT** decrypts Microsoft BASIC programs saved in protected format (preventing LISTing or EDITing). $39. FNB will supply updates for the cost of media and a "nominal handling charge." **(Reader Service No. 354)**

What? You don't know how to use CP/M? **TEACH/M** ("teach 'em," get it?), from Select, "helps frustrated users learn CP/M in ninety minutes." This disk for noncomputer professionals costs $75. **(Reader Service No. 364)**

Cheaper, and certainly more comprehensive, is a book, *Inside CP/M, A Guide for Users and Programmers*, published by Holt, Rinehart and Winston, and written by *DDJ* contributing editor Dave Cortesi. The $20, 575-page paperback contains two independent parts. The first is a comprehensive tutorial on the purchase, use, management and programming of a computer system with CP/M. The second part is a large reference section in which, according to the preface, "the answer to any question about the day-to-day use of CP/M can be found in seconds." **(Reader Service No. 449)**

UNERA, from Elliam Associates, unerases ERAsed CP/M files. UNE2 recovers the files, one at a time, or selectively from all erased files on a disk, and CON2 gives the status of erased files that are using the same disk space as active files (causing a space conflict). Recovering such files can clobber active files. The manual suggests steps for file recovery in this case. UNERA is available for $76.50 for all CP/M 2.2 systems, single- and double-density, floppy and hard disk with standard CP/M directories, and for single-density 1.4 systems, and can be patched to 1.4 double density. "Recovering the results of just one mistake," they tell us, "such as 'ERA *.BAS' when you meant 'ERA *.BAK,' can easily pay for the cost of these two programs." Sounds good. **(Reader Service No. 374)**

CPMPOWER, from Computing! (I love names with !'s in them), is a set of 45 (!) CP/M housekeeping commands. It recovers erased files, calculates checksums, tests disk and reserve files for bad blocks, lists file size in sector and kilobytes, loads disk files to any memory location, reads/writes any track and sector from/to memory, displays/substitutes hex, ASCII, and binary, dumps memory in formatted hex/ASCII, searches memory for ASCII or hex using wildcards, fills memory with hex, moves blocks of memory, lists disk parameters, copies files readily by assigning them numbers and permitting a copy over a range, as, files 7-23, 27-33, etc., chains to other .COM files, and has four user-configurable custom commands. All this, and a 60-page manual, too for $149! **(Reader Service No. 384)**

Another disk utility package for CP/M 2.x comes from ELEKTROKONSULT in Norway. They claim the product is unique because it automatically adapts to any diskette format. **DDUMP** examines and patches any byte on any sector, to study how CP/M allocates disk storage and to examine or repair a damaged disk and recover lost data. **DTEST** tests a disk for bad spots, locking bad sectors out from later use by CP/M by automatically collecting them in a "garbage" file. This reduces the likelihood of later disk crashes. **DUSER** permits access from one user area to programs/files on other user areas, without having to keep duplicate copies of files on the disk. **DDUP** duplicates disks independent of disk controller, drive, disk size, and format, and replaces bad sectors on the source disk with blank sectors on the destination, permitting recovery from damaged files. Now here's an anomaly. They too have an **UNERA** that recovers accidentally erased files (they even spell it the same, "ERAsed"). These utilities cost $29.95 each, or all five for $125. All five manuals alone, $20. Add $8 handling and shipping for each order, and specify format. **(Reader Service No. 394)**

For those doing development for the TI TMS7000 series microprocessors, Allen Ashley claims that his **SYSTEM-TMS7** is the only CP/M cross-assembler for these processors. The software system features a macroassembler with full macro and conditional assembly as well as the ability to chain a series of source files together during a single assembly, an interactive editor/assembler, a text editor, and offloading facilities. He also sells cross-assemblers for the National COP400 and 70-series, Intel 8048 and 8051, Zilog Z8, RCA 1802, AMI S2000, and Fairchild/Mostek F8/3870 processors. SYSTEM-TMS7 costs $150, including complete documentation and full user support by mail or phone. **(Reader Service No. 404)**

A 60K **CP/M System Upgrade Kit** for IMSAI costs $255 with CP/M, or $175 for just CP/M and $75 for just the kit. **(Reader Service No. 414)**

The **Catalog of Public Domain Software for CP/M**, Book 2, consisting of tables of contents and abstracts of files, and document files contained on diskette volumes published by the CP/M Users (sic, sans apostrophe) Group and the SIG/M (Special Interest Group/Microcomputers), is available from the New York Amateur Computer Club for $10. There is also, naturally enough, a Book 1. **(Reader Service No. 424)**

Here's a freebie from Digital Research: the **Technical Support Group** offers a set of CP/M 2.2 patches and application notes, "Using PIP with XSUB," "Auto Loading Programs," "Making Delete (Rubout) Work Like Backspace," "Improving the Control-S Function," and "Nested Submit Files." Request must be written and include the user's CP/M license number. CP/M users who have not sent in their registration cards can include them. Write to: CP/M 2.2 Application Notes, Technical Support, Digital Research, Box 579, Pacific Grove, CA 93950.

Buy Some Hardware

The **TI LCD Programmer** converts floating-point numbers between hexadecimal, decimal, and octal. There is a two's complement key, bit-by-bit logical operations, and fifteen sets of parentheses at each of four processing levels. Inmac sells it for $75 on a 45-day free trial basis, with one-year warranty against material defect and workmanship. How can you go wrong? **(Reader Service 426)**

Anyone who's had a crash from a spike on the line will appreciate Ico-Rally's **Surge Protector**. Sporting a 25 nanosecond response time, it can dissipate 950,000 watts at 100 microseconds, at its rated 15 amps and 120 volts AC, and costs a mere $39.95. **(Reader Service No. 434)**

If your only IBM product is a Selectric, **Electric Typing Fingers (ETF-80)**,

from Personal Micro Computers, can turn it into a printer. The unit sits atop the typewriter keyboard, which needs no modification (and thus the manufacturer's warranty remains in force), and plastic-tipped solenoids press the keys just as would a typist. Electric Typing Fingers runs on many popular computers, including the IBM PC, and operates the Selectric and other compatible typewriters. It needs no additional software, and costs $595. **(Reader Service No. 444)**

Osborne announced a **Double Density Upgrade** kit for the Osborne 1. It doubles storage to 184K, and reads and writes other formats, such as the Xerox 820. The system automatically recognizes different densities when diskettes are changed. Consisting of a printed circuit board, 2732A ROM, double-density system diskette, and user's manual, the kit costs $185 and is available at Osborne dealers. **(Reader Service No. 454)**

The Fox is Digital Microsystem's new DSC-3/F CP/M desktop computer, a 30-pound portable unit incorporating a nine-inch CRT, two 5-1/4-inch single- or double-density double-sided floppies (with a formatted capacity of 307.2K per drive), twelve programmable function keys, an interface for the company's local area network (HiNet) that can support 32 users and address up to 255, four RS-232C serial ports, two eight-bit bidirectional parallel ports with status lines, 64K of RAM and 1K of ROM. The processor is a Z-80A, providing an access speed of 250 nanoseconds, with no wait states. Programmable CRT control codes provide compatibility with other software drivers, so that the Fox can emulate any other terminal. The package includes business application software. Two word processors are offered, two data base management systems, budget and finance programs, and several programming languages, including BASIC, COBOL, PL/1, FORTRAN, and Pascal. It sounds like quite a system for $3995. **(Reader Service No. 459)**

Sell Some Software

Lifeboat Associates is offering a free publication, *Guidelines for Software Authors*. It addresses, among others, these questions: How should you prepare a software package for evaluation? What kind of documentation should you include with a submission? What about your legal rights? If your program is accepted, what next? **(Reader Service No. 464)**

Peachtree Software seeks quality, well-documented applications software for micros. **(Reader Service No. 474)**

PocketInfo asks authors of quality applications software for hand-held computers (TRS-80 pocket computer, Sharp PC-1211, and Casio FX-702P) to contact them for a copy of procedures and a non-disclosure agreement prior to submission. **(Reader Service No. 484)**

Atari has opened a regional center for the development and acquisition of software programs for home computers in the Boston area, at the Crimson Galleria, in Cambridge. This center is the second of its kind – the first is at corporate headquarters in Sunnyvale, California – and will "provide a place where qualified software developers can work with Atari computers, have access to our technical reference materials, and be able to work with our staff . . . who can answer their questions about our computers." The center will also house a facility for converting programs written for other computers to Atari format. Atari wishes to acquire software in the fields of entertainment, self-improvement, home management, education, and system software. In addition to developing their own software, marketing user-written software through the Atari Program Exchange, and working with other established software publishing houses, especially in the educational area, Atari encourages third-party developers. They plan other such centers in areas with high concentrations of programmers and users. **(Reader Service No. 494)**

. . . Well, at Least Join a User Group

CP/M user groups include the **New York Amateur Computer Club**, Box 106, Church Street Station, New York, NY 10008; the **CP/M Users Group**, 1651 Third Avenue, New York, NY 10028, and **SIG/M** (Special Interest Group/Microcomputers), Box 97, Iselin, NJ 08830.

CP/M users will also be interested in **PicoNet**, which meets monthly in Dysan in Santa Clara, CA. $20 a year gains access to a large library of member-contributed software and a newsletter. Box 2070, Sunnyvale, CA 94087.

For those who thought the Sorcerer had died (including me; I have one), the **Sorcerer's Apprentice Users Group** publishes a fact-filled 24-page newsletter with information about Exidy machines that can be found nowhere else, maintains a Sorcerer CBBS system in the Detroit area (313-535-9186 ringback), and makes a lot of wonderful software available. Membership is $18 a year, or $24 for the newsletter sent first class. Sorcerer's Apprentice, Box 33, Madison Heights, MI 48071; (313) 286-9265.

North Star owners can join the **International North Star Users Association** for $20 a year. Box 1318, Antioch, CA 94509.

Membership dues in the **MUMPS Users' Group of North America** (MUG), are $40 a year, and include a subscription to MUG-Quarterly, discounts on MUG publications, and voting privileges. MUMPS Users' Groups, Box 37247, Washington, DC 20013. (301) 779-6555. (They must be very professional. They seem to be the only ones who know where the apostrophe belongs!)

DDJ

Volkswriter

Lifetree Software
$195
Circle No. 489 on Reader Service Card
Reviewed by Robert K. Funkhouser

VOLKSWRITER, a product of Lifetree Software, is designed specifically for the IBM Personal Computer. It makes extensive use of the cursor control keys, function keys and other special keys of the PC keyboard to make word processing an easy task. VOLKSWRITER is written in IBM Pascal and runs under the DOS operating system.

The software was tested on a single-drive system with 64K RAM. Using a single drive posed no problems, just the usual hassle of switching diskettes to save and retrieve files. The amount of RAM, however, proved a serious limitation.

VOLKSWRITER is advertised as a high-performance word processor. Indeed, most commands and functions are performed extremely fast. However, on a 64K machine you are left with just 10K memory for editing. This is a serious limitation, as 10K is just too small. The VOLKSWRITER documentation states that up to 128K can be used, and strongly recommends you have that much memory. On a 128K machine, you will have 60K available to hold edited data.

The documentation is good. It is easy to read and is structured for quick reference. Three tutorial files come on the program diskette. These guide you through the use of function keys and other special keyboard keys used in editing. It requires only a couple of hours, using the tutorials and the manual, to develop proficiency with all of VOLKSWRITER's features. Learning to use it was quick, easy and fun.

There are two main menus: an Editor menu and a Printer menu. The Editor menu contains the full range of editing functions and minor printing capabilities. The Printer menu is used to print documents but also has minor editing functions. Thus, you would normally start with the Editor menu, edit and save a file, then call the Printer menu to create a print format and print the file.

Editing

Under the Editor menu the following can be done:

Display files on diskette
Retrieve a file for editing
Edit a file
Store a file on diskette
Select a print format for use
Clear memory for editing a new file
Rename files
Delete files
Select the Printer Menu
Display HELP text
Exit VOLKSWRITER

The main purpose of a word processor is to manipulate text, and the VOLKSWRITER is up to the task. The editing functions make extensive use of the special keys on the keyboard. The following shows the editing functions and the keyboard keys associated with them:

Cursor and text movement:

Home	moves cursor to the upper left corner of the screen
End	moves cursor to the lower left corner of the screen
→	moves cursor to the right
←	moves cursor to the left
↑	moves cursor up; if the cursor is at the top of the screen it forces upward scrolling
↓	moves cursor down; if the cursor is at the bottom of the screen it forces downward scrolling
Ctrl + →	moves cursor to the next word
Ctrl + ←	moves cursor to the previous word
Tab	tabs to preset columns
Ctrl + Home	moves the cursor to the first character of text
Ctrl + End	moves the cursor to the last character of text
Pg Up	scrolls upward one-half screen
Pg Dn	scrolls downward one-half screen
Ctrl + Pg Up	scrolls upward one screen
Ctrl + Pg Dn	scrolls downward one screen

Inserting text:

Ins	inserts characters; if insertion pushes the line past the right margin, a new line is started
F3	inserts a new line
Alt + F3	inserts a section symbol marking the end of a page (used in printing)
Alt + F2	inserts text from another file

Deleting text:

Del	deletes characters
F4	deletes a word: if the cursor is in the middle of a word only the text to the right of the cursor is deleted
Alt + F4	deletes characters from the cursor position to the end of the line; if the cursor is in column 1 then the entire line is deleted
Alt + F8	deletes a block of text

All other functions:

F5	defines the beginning of a block
F6	defines the end of a block
Alt + F5	moves a block of text
Alt + F6	copies a block of text
Alt + F8	deletes a block of text
F1	Help; displays function key settings
Alt + F1	Prints text from cursor position to the end of the file
F2	writes the entire text or portions of the text to a file
F7	Find; finds a specified string and places the cursor at the string
Alt + F7	Replace; replaces one string with another. This may be done for one, several or all occurrences of the string.
F8	Reformat; reformats text until an end of paragraph or end of page symbol is found.
F9	used to set tab positions and the right margin
Alt + F9	centers line of text within margins
F10	Reset; cancels block definitions, stops printer, resets F7 strings, etc.
Alt + F10	Exit; return to main menu.

All of the above functions proved easy to use. Perhaps as significant, these functions are easy to remember. The uses of the cursor control keys relate well to the names of the keys. After using VOLKSWRITER for about thirty minutes, the uses of those keys were well in mind. Recalling the purposes of the function keys is a snap, simply enter F1 to see the settings.

Throughout all menus, VOLKSWRITER displays a status line at the bottom of the screen. This line shows the name of the file being edited, and the status of a Retrieve, Store, Print, etc., activity. The status line also has a "gas gauge" which shows what percentage of memory (left after VOLKSWRITER was loaded) is available for use. When this percentage falls below 10, the percentage display begins to flash. It's meant to get

your attention and it works. When the percentage gets to zero (or near zero) the speaker is beeped ten times in a shrill tone and you are instructed to immediately save the file. This feature, coupled with VOLKSWRITER's insistence (almost to the point of being annoying) that you respond to a Yes/No prompt before taking any action that would erase memory, makes it very hard to accidentally lose what you have edited.

Printing

VOLKSWRITER uses "print formats" which define the characteristics of the printout. How many lines per page, what line to put page numbers (if any) on, what page number to begin with, margin specifications, number of extra spaces (if any) between lines, along with several other options. Formats are also used to specify left and right justification. Left and right justification is performed on the screen by using the F8 (paragraph reformat) key.

Some of the options pertain strictly to the IBM (or Epson) dot matrix printer, such as dark printing, double strike printing, 40, 66, 80 or 132 characters per line. When used with the NEC Spinwriter printer, VOLKSWRITER issues codes for inter-line spacing and horizontal motion index at the start of executing the Print command.

A format may be set for a single use or you may save the format as a file for later use. This technique makes it easy to set up standard formats for particular kinds of documents as well as the creation of custom formats for a single use.

The Printer menu is like the Editor menu except it allows you to set up a new format, print, and return to the Editor menu. While you may do editing through this menu, not all of the editing functions are available In particular, the block functions, block moves, copies, etc. are not available. Thus, editing through this menu would normally be done for minor changes only, such as last minute changes before printing.

Printing may also be done by placing . .PRINT and . .FORMAT commands in the text. Here's an example:

```
. .FORMATCONTENTS.FMT
. .PRINTCONTENTS.TEX
. .PAUSEInsert new forms
```

The first line will call in the print format CONTENTS.FMT and the second line will cause CONTENTS.TEX to be printed using the CONTENTS.FMT format. The . .PAUSE will print the message "Insert new forms" and cause a system wait until any key is pressed. By using this technique you can print several files at one time or call in "boilerplate" text files at specific points. This is also the way to print a file or files that are too large to fit into memory.

Shortcomings

The only problem encountered was when doing a reformat of a paragraph (the F8 key) which contained a continuous string of characters longer than one line. I was playing with the command (one would normally not have such a character string in a paragraph) when the problem occurred. An "Lstring too long . . ." error was displayed, and VOLKSWRITER went down. I restarted it by entering VW (normal startup) and, as expected, the file I was editing was gone. Although this situation might never occur in "real use," It's always disappointing when software fails and information is lost.

Another complaint regards the messages displayed when it is necessary to switch diskettes. These are really DOS messages and they are often placed in the middle of menu displays, making them difficult to read and often confusing. Once you get used to this, however, it presents no functional problems, just some messy messages. Two-drive systems should not have this distractive situation.

If you replace all occurrences of a given string with a longer string you wil find that all paragraphs which have been changed will need reformatting (F8). You have to page through your text and press F8 for each paragraph that needs reformatting. It would nice if there were a "global reformat" function to speed the process.

Summary

Performance was very good, albeit at the expense of memory. VOLKSWRITER sells for $195, but if you have just 64K you should add the price of another 64K. I had not planned to expand the memory on my machine, but after using VOLKSWRITER, and seeing the dramatic increase in my writing speed, I'm not complaining too loudly about having to get more memory. I would probably need it or want it sooner or later, anyway. VOLKSWRITER itself will use 14K more if you have 128K, which leads me to believe it will require even fewer trips to disk, thus improving response time when moving between menus.

My overall impression of VOLKSWRITER is very positive. It allows for excellent control of the printout, has excellent protection against data loss (discounting the odd reformat crash) and fully uses the IBM-PC keyboard. It also allows the fonts and darkness options of the IBM matrix printer to be easily used. It is a good piece of software tailored to match a good machine.

Write-On

Spiecher Systems
Published by Datamost
$129.95
Circle No. 499 on Reader Service Card
Reviewed by Jerry Crimmins

Write-On! is a converted Applesoft program from Speicher Systems. Being written in Microsoft Basic for the Apple Computer probably made this program easy to convert to Microsoft for the IBM. It also could be why Ctrl-C is used as an imbedded character to center a line (this can cause an unintentional break, but text is not lost). This may explain why the editor in the program runs slower than other word processors. But the experience gained from Betsy Speicher's previous Apple adventure obviously carried over to this program.

Unlike another widely advertised word processor for the IBM Personal Computer, Write-On! will read, write, or merge any standard IBM Personal Computer ASCII file. For some people, that alone may be reason enough to buy Write-On! There are many good things about this word processing package. It has an impressive array of formatting and printer commands that allow the user to easily re-arrange text, and to add numbers and titles to pages or entire chapters with only a few keystrokes. All margins are adjustable, 20 pre-set headers are available that can be cleared and redefined during an edit session, and 9 tab settings can be cleared and reset. Block transfers in the editor are easy to learn and do, global changes are easy, and text can be formatted and previewed on the screen before printing. In short Write-On! does anything that those other word processors do.

In one instance, a form letter can be created that will prompt a typist to enter the appropriate text in the appropriate place. Similarly, a form letter will read a data file containing the appropriate kinds of data input, while it prints. Combine this capability with disk image files (text stored on disk in a formatted version) to store form letters, then make a mailing list data file by using the included data file writer, and print form letters automatically. One of the on-disk examples is a form letter (and another is an invoice billing) set to read data files.

Write-On! is good, but it is not perfect. It doesn't support the underlining or overstrike capability of the IBM printer (Epson).

A touch-typist might never realize that there are better editors than the one provided with Write-On! But a non-typist (like myself) may find the current line pointer a little difficult to manipulate. It is done by entering commands U (up) or D (down) followed by the number of the lines you wish to move. Perhaps a scrolled pointer, or the addition of line numbers to the text would be sufficient. With neither option, it is difficult to place the pointer on the right line, especially when seeking text that is off the screen.

There are some good parts to the editor. It does make use of the cursor control keys on the number keypad (within one 255 character line). The "Ins" and "Del" keys, the "End" and "Ctrl End" (delete line from cursor to end) keys work in the edit command mode by pushing the return key. This editor is quite sufficient and will do the job for anyone. All the commonly used control characters are in the typewriter area of the keyboard, which is good for the touch-typist.

The documentation that comes with Write-On! is slow and easy, but it does contain everything. The biggest need is for more examples of unformatted text in the manual, where it can be referred to while typing in a file. The manual does contain all the necessary information to teach a beginner how to use the programs, which is more than most manuals.

There is an error message index in the back and a handy reference card (prompter) with commands, defaults, and syntax.

The best part of the documentation is the files on the Write-On! diskette that illustrate the unformatted, versus the formatted text. These are excellent examples. In addition to the previously mentioned form letter is a billing invoice, a meat loaf recipe, and an outline of the entire history of Middle Earth.

Write-On! is as powerful as other microcomputer word processors, and it costs less. Compared to Word Star or Easy Writer 1.11, Write-On! is better prepared to be all things for all users. True, the editor is not a machine language routine with fast vertical and horizontal scroll; but it does work and doesn't introduce lots of new keys to touch-typists.

The power of the imbedded printer commands, and the disk file commands allows Write-On! to fill commercial uses right away.

After using this software to write some letters (and of course this review), I had some questions for Datamost and Betsy Spiecher – I had trouble getting files to and from drive b. No problem, I was told, use b:File.Spc (no space between ":" and "File").

Another subject we discussed was backup copies; everyone talks about copying. If a program is not backed up an investment may be wasted, or a customer may experience an unacceptable delay. Write-On! is copyable, and this recommends the product over those programs that can't be backed up. Another good thing about Basic is that it is easy to modify. The company says that an improved editor is on the way, with an even better manual (the entire manual was set using Write-On!). Datamost was very helpful, and it was a pleasure to talk with the author of the program.

I obviously appreciate this software. It is a writing utility, it handles files and records, it handles the printer, and it handles the editing. It is not fast and it doesn't scroll. But there is no other one package I could find which does everything that Write-On! does, and especially not for the IBM Personal computer. After a quick update to support underlining on the IBM/Epson printers (yes, the Epson 100 also), and a little souping up on the editor, it will be difficult to find a better word processor for any micro.

DDJ

Small-C for the 9900

by Matthew Halfant

Matthew Halfant, Bedford Computer Corp., Tirrell Hill Rd., Bedford, NH 03102.

In number 54 of *Dr. Dobb's*, Ron Cain published his subset C compiler, written in its own language. For $15 I obtained from The Code Works a working version (CP/M Version N – April 1, 1981) of this compiler, complete with a runtime interface to CP/M. I proceeded to rewrite the code generator part to produce assembly code for my 9900, an earlier acquisition without disks of its own but having access to disks via an interface to my CP/M machine. Thus I now write C programs which I cross-compile to run on the 9900.

The model of the machine addressed by Ron's compiler involves 16-bit primary and secondary registers, and a stack. Since the 9900 is a 16-bit machine, I thought at first that the conversion would be straightforward. It was not. One very minor problem is that the 9900 does not use a stack for return addresses in making subroutine calls. More serious is the fact that it does not want to access word (i.e., 16-bit) operands unless they are aligned on word boundaries.

The solution to the first problem is to create a stack: I used general register 0 (denoted R0 in what follows) as the stack pointer. The 8080 instruction

```
CALL SUBR
```

is replaced here by

```
BL   @SUBCAL
DATA SUBR
```

where SUBCAL is a single routine posted in the preamble of the generated code, which branches to SUBR after saving the return address on the stack. The code for SUBCAL is

```
SUBCAL MOV  *R11+,R3
       DECT R0
       MOV  R11,*R0
       B    *R3
```

The return from a subroutine is effected by B @SUBRET, where SUBRET is simply

```
SUBRET MOV  *R0+,R3,
       B    *R3
```

As mentioned above, a far more serious problem concerned operand alignment. Global variables are given reserved space in RAM, so that, for example, a program beginning

```
int  x;
char c;
int  z;
main()
{   ...
```

would produce, in the assembly listing

```
ZX   BSS  2
ZC   BSS  1
ZZ   BSS  2
```

(the prefix z – Code Works uses qz – prevents conflict with assembler keywords). Since c is a char, it will be accessed with a MOVB instruction (MOVe Byte), not sensitive to alignment. But z will be accessed as a word, and we see that it may not be aligned as things stand.

A similar thing occurs with local variables, for which space is reserved on the stack. In my first pass at rewriting the code generator, I compromised the 9900 by forcing byte moves, rather than risk word moves that could not be guaranteed to work. Except for some register-to-register operations, I was cutting a poor figure compared to the 8080 for which the original was written. The size of the generated code was excessive. For example, the 8080 pushes its primary register (HL) on the stack with

```
PUSH H
```

and I was achieving the same thing with:

```
DEC   R0
MOVB  R1,*R0
SWPB  R1
DEC   R0
MOVB  R1,*R0
SWPB  R1
```

Not ideal.

The correct approach, which I subsequently adopted and advocate here, is to modify the compiler itself so that alignment is preserved. This is done very simply as follows. The routine dumpglbs(), from ccl, generates the storage definition statements illustrated above (see Figure 1). A test is made to see if an odd number of bytes is being reserved; if so, that number is incremented by 1. The inserted line if(j&l) j++; does the job.

For the situation with local variables, the identical approach is applied, this time to decIloc(typ), which is the last function in ccl. The inserted line is as before, here using k in place of j (see Figure 2).

With these changes, I was able to rewrite the code generator cc8 as shown in Listing I. In adapting cc8 to other 16-bit machines, it may be well to incorporate the above changes even if the machine will allow unaligned accesses; the code is bound to run faster and the cost in additional storage is small.

By the way, some minor additional alterations were needed for the case of the 9900, and I mention them here for

Figure 1

```
dumpglbs()
     {
     int j;
     if(glbflag==0)return;     /* don't if user said no */
     cptr=startglb;
     while(cptr<glbptr)
          {if(cptr[ident]!=function)
               /* do if anything but function */
               {outname(cptr);col();
                    /* output name as label... */
               defstorage();  /* define storage */
               j=((cptr[offset]&255)+
                    ((cptr[offset+1]&255)<<8));
                         /* calc # bytes */
               if((cptr[type]==cint)|
                    (cptr[ident]==pointer))
                    j=j+j;
               if(j&1) j++;    /* 9900 addition: reserve even */
               outdec(j);      /* number of bytes */
               nl();
               }
          cptr=cptr+symsiz;
          }
     }
```

those who want to make use of what I've done.

1. Change #define litabsz 2000 before attempting any additions to the compiler. I use 4000.
2. In main(), I invert the order of calling dumplits() and dumpglbs(). Thus the string literals are dumped last, using the BYTE pseudo-op. Doing it the other way would require insertion of an EVEN pseudo-op between the two.
3. The function col(), which prints a colon, is rendered a no-op. This is used only to add a colon following statement labels, which the 9900 assembler does not want.
4. The statements #define asmpref 7, #define asmsuff 7, are modified to use constants 5 and 0, respectively. The routine outname(sname), from cc8, prefixes only "z" instead of "qz" as in the original, which results in maximum symbolic name lengths of 6. This unfortunate restriction is imposed by the 9900 assembler.

In addition to the code generator, it is necessary to write a runtime library containing the functions getchar(), putchar(c), gets(buff), puts(string), fopen(name,mode), fclose(ptr), getc(ptr), putc(c,ptr), and cpm(bc,de). These are described in the 11-page manual supplied by The Code Works with their diskette. Adherence to their definitions guarantees that the compiler can be bootstrapped to the target machine.

(The function cpm() is used only once in The Code Works version, and the reference can be eliminated. That makes it unnecessary to include the function with the run-time library.) ▶▶J

(Listing begins on page 68)

Figure 2

```
declloc(typ)        /* typ is cchar or cint */
     int typ;
     {
     int k,j;char sname[namesize];
     while(1)
          {while(1)
               {if(endst())return;
               if(match("*"))
                    j=pointer;
                    else j=variable;
               if (symname(sname)==0)
                    illname();
               if(findloc(sname))
                    multidef(sname);
               if (match("["))
                    {k=needsub();
                    if(k)
                         {j=array;
                         if(typ==cint)k=k+k;
                         }
                    else
                         {j=pointer;
                         k=2;
                         }
                    }
               else
                    if((typ==cchar)
                         &(j!=pointer))
                         k=1;else k=2;
               if(k&1) k++; /* for 9900: keep stack word-aligned */
               /* change machine stack */
               Zsp=modstk(Zsp-k);
               addloc(sname,j,typ,Zsp);
               break;
               }
          if (match(",")==0) return;
          }
     }
```

CP/M Exchange

(Continued from page 9)

work to do that does not require any printer output and I'd like to keep it "OFF".

Sincerely,
A Dedicated Reader

Without your input this feature will not continue. Figure out what you have learned that will benefit others and/or problems you have yet to solve. Send them to the CP/M Exchange. I think I can safely say that whatever interests you will also interest other readers of this column. Information exchange ends when the data is acknowledged as received correctly. Information exchange begins when you ask a question or share an idea.

▶▶J

Small-C for 9900 (Text begins on page 66)

```
                CODE GENERATOR FOR THE 9900

/*   >>>>>> start of cc8 <<<<<<<   */

/* Begin a comment line for the assembler */
comment()
{    outbyte('*');
}

/* Put out assembler info before any code is generated */
header()
{    comment();
     outstr(BANNER);
     nl();
     comment();
     outstr(AUTHOR);
     nl();
     comment();
     outstr(VERSION);
     nl();
     comment();

nl();
if(mainflg){        /* do stuff needed for first */
     outstr("TOPMEM EQU  >8800"); /* initial stack pointer */
     nl();
     ol("DREG");
     ol("AORG  >100");
     ol("JMP   START");
     outstr("SUBCAL MOV  *R11+,R3");
     nl();
     ol("DECT  R0");
     ol("MOV   R11,*R0");     /* push return address on stack */
     ol("B     *R3");
     outstr("SUBRET MOV  *R0+,R3");     /* pop address from stack */
     nl();
     ol("B     *R3");
     outstr("START  LI   R0,TOPMEM");   /* R0=stack pointer */
     nl();
     zcall("main"); /* call the code generated by small-c */
     zcall("exit"); /* do an exit       gtf 7/16/80 */
     }
}

/* Print any assembler stuff needed after all code */
trailer()
{
     ol("EVEN");
     ol("DATA  -1");     /* 9900 assembler seems to need this */
     ol("END");
     comment();
     outstr(" --- End of Compilation ---");
     nl();
}

/* Print out a name such that it won't annoy the assembler */
/*   (by matching anything reserved, like opcodes.) */
/*   gtf 4/7/80 */
outname(sname)
char *sname;
{    int len, i,j;

     outasm("z");
     len = strlen(sname);
     if(len>(asmpref+asmsuff)){
          i = asmpref;
          len = len-asmpref-asmsuff;
          while(i-- > 0)
               outbyte(raise(*sname++));
          while(len-- > 0)
               sname++;
          while(*sname)
               outbyte(raise(*sname++));
          }
     else outasm(sname);
/* end outname */}

/* Fetch a static memory cell into the primary register */
getmem(sym)
     char *sym;
{    if((sym[ident]!=pointer)&(sym[type]==cchar))
          {ot("MOVB @");
          outname(sym+name);
          outstr(",R1    ;getmem(),char");
          nl();
          ol("SRA   R1,8");
          }
     else
          {ot("MOV  @");
          outname(sym+name);
          outstr(",R1    ;getmem(),int");
          nl();
          }
     }

/* Fetch the address of the specified symbol */
/*   into the primary register */
getloc(sym)
     char *sym;
{    immed();
     outdec(((sym[offset]&255)+
          ((sym[offset+1]&255)<<8))-
          Zsp);
     outstr("  ;getloc");
     nl();
     ol("A     R0,R1");  /* add stack pointer to primary */
     }

/* Store the primary register into the specified */
/*   static memory cell */
putmem(sym)
     char *sym;
{    if((sym[ident]!=pointer)&(sym[type]==cchar))
          {ol("SWPB R1");
          ot("MOVB  R1,@");
          outname(sym+name);
          outstr("  ;putmem(),char");
          nl();
          ol("SWPB  R1");
          }
     else {ot("MOV  R1,@");
          outname(sym+name);
          outstr("  ;putmem(),int");
          nl();
          }
     }

/* Store the specified object type in the primary register */
/*   at the address on the top of the stack */
putstk(typeobj)
char typeobj;
{    zpop();

     if(typeobj==cint)
          {ol("MOV  R1,*R2    ;putstk(),int");
               /* primary stored indirect through sec */
          }
     else {    ol("SWPB  R1   ;putstk(),char");
          ol("MOVB  R1,*R2");
          ol("SWPB  R1");
          }
     }

/* Fetch the specified object type indirect through the */
/*   primary register into the primary register */
indirect(typeobj)
     char typeobj;
{    if(typeobj==cchar)
          {ol("MOVB *R1,R1    ;indirect(),char");
          ol("SRA   R1,8");
          }
     else {ol("MOV  *R1,R1    ;indirect(),int");
          }
}

/* Swap the primary and secondary registers */
swap()
{    ol("MOV   R1,R3     ;swap()");
     ol("MOV   R2,R1");
     ol("MOV   R3,R2");
}

/* Print partial instruction to get an immediate value */
/*   into the primary register */
immed()
{    ot("LI    R1,");
}

/* Push the primary register onto the stack */
zpush()
{    ol("DECT  R0   ;zpush()");
```

```
    ol("MOV   R1,*R0");
    Zsp=Zsp-2;
}

/* Pop the top of the stack into the secondary register */
zpop()
{   ol("MOV   *R0+,R2   ;zpop()");
    Zsp=Zsp+2;
}

/* Swap the primary register and the top of the stack */
swapstk()
{   ol("MOV   R1,R3     ;swapstk()");
    ol("MOV   *R0,R1");
    ol("MOV   R3,*R0");
}



/* Call the specified subroutine name */
zcall(sname)
    char *sname;
{   ol("BL    @SUBCAL");
    defword();
    outname(sname);
    nl();
}

/* Call a run-time library routine */
callrts(sname)
char *sname;
{
    ot("BL    @"); /* runtime routines save R11 if need be */
    outasm(sname);
    nl();
/*end callrts*/}

/* Return from subroutine */
zret()
{   ol("B     @SUBRET");
}



/* Perform subroutine call to value on top of stack */
callstk()
{   ol("MOV   *R0,R3    ;callstk()");
    ol("LI    R1,$+6");
    ol("MOV   R1,*R0");
    ol("B     *R3");
    Zsp=Zsp+2; /* corrected 5 May 81 rj */
    }

/* Jump to specified internal label number */
jump(label)
    int label;
{   ot("B     @");
    printlabel(label);
    outstr("  ;jump()");
    nl();
    }



/* Test the primary register and jump if false to label */
testjump(label)
    int label;
{   ol("MOV   R1,R1     ;testjump()");
    ol("JNE   $+4");
    ot("B     @");
    printlabel(label);
    nl();
    }

/* Print pseudo-op to define a byte */
defbyte()
{   ot("BYTE  ");
}


/*Print pseudo-op to define storage */
defstorage()
{   ot("BSS   ");
}

/* Print pseudo-op to define a word */
defword()
{   ot("DATA  ");
}
```

```
/* Modify the stack pointer to the new value indicated */
modstk(newsp)
    int newsp;
 {  int k;
    k=newsp-Zsp;
    if(k==0)return newsp;
    ot("AI    R0,");
    outdec(k);
    outstr("  ;modstk()");
    nl();



    return newsp;
}

/* Double the primary register */
doublereg()
{   ol("SLA   R1,1 ;doublereg()");
}

/* Add the primary and secondary registers */
/*   (results in primary) */
zadd()
{   ol("A     R2,R1     ;zadd()");
}

/* Subtract the primary register from the secondary */
/*   (results in primary) */
zsub()
{   ol("S     R2,R1     ;zsub()");
    ol("NEG   R1");
}



/* Multiply the primary and secondary registers */
/*   (results in primary */
mult()
{   ol("CLR   R5   ;mult()");          /* signed multiply */
    ol("MOV   R2,R3");
    ol("JGT   $+4");
    ol("NEG   R3");
    ol("INC   R5");
    ol("MOV   R1,R1");
    ol("JGT   $+4");
    ol("NEG   R1");
    ol("INC   R5");
    ol("MPY   R1,R3");
    ol("MOV   R4,R1");
    ol("ANDI  R5,1");
    ol("JEQ   $+2");
    ol("NEG   R1");
}



/* Divide the secondary register by the primary */
/*   (quotient in primary, remainder in secondary) */
div()
{   ol("CLR   R5   ;div()");      /* signed divide */
    ol("MOV   R2,R4");
    ol("JGT   $+4");
    ol("NEG   R4");
    ol("INC   R5");
    ol("CLR   R3");
    ol("MOV   R1,R1");
    ol("JGT   $+4");
    ol("NEG   R1");
    ol("INC   R5");
    ol("DIV   R1,R3");
    ol("MOV   R3,R1");
    ol("MOV   R4,R2");
    ol("ANDI  R5,1");
    ol("JEQ   $+4");
    ol("NEG   R1");
    ol("NEG   R2");
}

/* Compute remainder (mod) of secondary register divided */
/*   by the primary */
/*   (remainder in primary, quotient in secondary) */
zmod()
{   div();
    swap();
    }
```

(Continued on next page)

Small-C for 9900

(Listing continued, text begins on page 66)

```
/* Inclusive 'or' the primary and the secondary registers */
/*   (results in primary) */
zor()
     {ol("SOC  R2,R1     ;zor()");}

/* Exclusive 'or' the primary and seconday registers */
/*   (results in primary) */
zxor()
     {ol("XOR  R2,R1     ;zxor()");}

/* 'And' the primary and secondary registers */
/*   (results in primary) */
zand()
     {ol("MOV  R2,R3     ;zand()");
     ol("INV   R3");
     ol("SZC   R3,R1");
     }

/* Arithmetic shift right the secondary register number of */
/*   times in primary (results in primary) */
asr()
     {ol("MOV  R0,R3     ;asr()");
     ol("MOV   R1,R0");
     ol("MOV   R2,R4");
     ol("SRA   R2,0");   /* shift count in R0 */
     ol("MOV   R2,R1");
     ol("MOV   R4,R2");  /* restore secondary */
     ol("MOV   R3,R0");
     }

/* Arithmetic left shift the secondary register number of */
/*   times in primary (results in primary) */
asl()
     {ol("MOV  R0,R3     ;asl()");
     ol("MOV   R1,R0");
     ol("MOV   R2,R4");
     ol("SLA   R2,0");   /* shift count in R0 */
     ol("MOV   R2,R1");
     ol("MOV   R4,R2");  /* restore secondary */
     ol("MOV   R3,R0");
     }

/* Form two's complement of primary register */
neg()
     {ol("NEG  R1   ;neg()");}

/* Form one's complement of primary register */
com()
     {ol("INV  R1   ;com()");}

/* Increment the primary register by one */
inc()
     {ol("INC  R1   ;inc()");}

/* Decrement the primary register by one */
dec()
     {ol("DEC  R1   ;dec()");}

/* Following are the conditional operators */
/* They compare the secondary register against the primary */
/* and put a literal 1 in the primary if the condition is */
/* true, otherwise they clear the primary register */

/* Test for equal */
zeq()
     {ol("MOV  R1,R3     ;zeq()");
     ol("CLR   R1");
     ol("C     R2,R3");
     ol("JNE   $+2");
     ol("INC   R1");
     }

/* Test for not equal */
zne()
     {ol("MOV  R1,R3     ;zne()");
     ol("CLR   R1");
     ol("C     R2,R3");
     ol("JEQ   $+2");
     ol("INC   R1");
     }

/* Test for less than (signed) */
zlt()
     {ol("MOV  R1,R3     ;zlt()");
     ol("LI    R1,1");
     ol("C     R2,R3");
     ol("JLT   $+2");
     ol("CLR   R1");
     }

/* Test for less than or equal to (signed) */
zle()
     {ol("MOV  R1,R3     ;zle()");
     ol("CLR   R1");
     ol("C     R2,R3");
     ol("JGT   $+2");
     ol("INC   R1");
     }

/* Test for greater than (signed) */
zgt()
     {ol("MOV  R1,R3     ;zgt()");
     ol("LI    R1,1");
     ol("C     R2,R3");
     ol("JGT   $+2");
     ol("CLR   R1");
     }

/* Test for greater than or equal to (signed) */
zge()
     {ol("MOV  R1,R3     ;zge()");
     ol("CLR   R1");
     ol("C     R2,R3");
     ol("JLT   $+2");
     ol("INC   R1");
     }

/* Test for less than (unsigned) */
ult()
     {ol("MOV  R1,R3     ;ult()");
     ol("LI    R1,1");
     ol("C     R2,R3");
     ol("JL    $+2");
     ol("CLR   R1");
     }



/* Test for less than or equal to (unsigned) */
ule()
     {ol("MOV  R1,R3     ;ule()");
     ol("CLR   R1");
     ol("C     R2,R3");
     ol("JH    $+2");
     ol("INC   R1");
     }

/* Test for greater than (unsigned) */
ugt()
     {ol("MOV  R1,R3     ;ugt()");
     ol("LI    R1,1");
     ol("C     R2,R3");
     ol("JH    $+2");
     ol("CLR   R1");
     }

/* Test for greater than or equal to (unsigned) */
uge()
     {ol("MOV  R1,R3     ;uge()");
     ol("CLR   R1");
     ol("C     R2,R3");
     ol("JL    $+2");
     ol("INC   R1");
     }

/*   <<<<<  End of small-c compiler     >>>>>     */
```

(End Listing)

(Letters)
(Continued from page 7)

from 1981 won't run on my 1982 p-machine, which in turn is reluctant to accept a new CPU. Neither p-machine will run programs written for the Apple, in spite of a great deal of hype about machine-independence. There must be a lesson here somewhere.

Sincerely,
D. C. Sessions
4649 E. Pueblo Avenue
Phoenix, AZ 85040

EPROM for Epson

Dear Marlin,

Thank you for publishing a number of comments and a printer driver for the Epson MX-80 printer in the November and December 1982 issues. I would like to make a few comments, and also describe some features which have not been discussed.

In the Clinic in November, there is some confusion as to the code used to switch off double-width printing. The code given in the Epson manual is DC4, or ctrl-T, and not SI or SO.

There is a feature in the printer which has not been mentioned, namely the possibility of printing Japanese characters instead of the TRS-80 graphics set. Presumably, not many users need this character set, and this is probably the reason why the escape codes 4 and 5 are not documented in the manual. If the Japanese characters are selected with DIP switch 1, pin 7, then ESC 5 can be used to switch to graphics characters, and ESC 4 to switch back to Japanese. These codes are ignored if DIP switch 1 selects graphics.

For those users who have access to an EPROM programmer, the Japanese characters can be substituted by 63 other characters; in my printer I have chosen to implement all the Greek lower-case characters (for mathematical and electronics texts), and a number of other useful special characters. Since I use graphics most, I have also changed ESC 4 and ESC 5 so they can be used irrespective of DIP switch 1.

To implement these features and a number of others, I have manually disassembled all the 4K of Intel 8049 instructions controlling the printer. In Denmark, for example, 12-inch paper is standard rather than the 11-inch kind found in the U.S. It was therefore logical to preset the forms length to 72 lines instead of 66 lines, and to allow page lengths up to 127 lines.

If Gary Novosielski has a printer which can be set to more than 66 lines per page, then there must be several versions of the ROM. The one I have (from about October '80) has the restriction mentioned in the manual, that the maximum is 66 lines per page. This limit seems ridiculous since it apparently serves no practical purpose to limit page size arbitrarily.

The graphics capabilities of the printer have not been mentioned at all, and obviously they are not as impressive as those available with the Graftrax ROMs or the MX-100. However, I have found them adequate for most of my needs, especially when printed in condensed form. In this format, each pixel is almost exactly twice as high as it is wide (the ratio is actually 18:34), and thus a square pixel can be obtained by using two adjacent horizontal pixels.

Unfortunately, in condensed mode two extra dot positions are inserted between each character, and therefore there is a noticeable space between every second pixel. This makes the graphics seem discontinuous. The alternative would be to remove these dot positions in the ROM, but this would change the pitch for condensed print as well. Actually, the condensed mode does not print 16.5 characters per inch, but 17 characters per inch. Try measuring a line of 132 characters; it should be almost exactly 7.7 inches rather than 8 inches long. Perhaps the original intention was to print 136 characters in compressed mode. Removing the two dots would give 20 pitch, or 160 characters per line, with the same character size as now, but with less space between the characters.

Another possibility would be to make one of the two horizontal pixels per character wider than the other one, but this may give some unbalanced graphics. Someday when I have the time I will try to implement this option.

From the above it should be obvious that Gary Novosielski is not quite correct when he states that the print head speed is reduced by a factor of 1.65 in compressed mode. It is reduced to half speed, the same as for emphasized printing. Dot spacing in emphasized mode is the same as in compressed mode, and the print head has only two speeds rather than three.

A useful feature which has not been incorporated into the printer is a reset switch. On the parallel interface there is an INIT line which does a hardware reset of the processor, but this is not implemented on my computer. It is annoying having to power off the printer to reset all the internal memory, and quite often it seems to disturb the computer as well when switching on or off.

Actually, it is simple to install a reset switch. I have chosen to install a push-button reset switch just behind the power switch. One lead is connected to the INIT line in the interface, the other is connected to ground. The easiest position to find these may be on resistor R15 and zener diode ZD5. These are located to the right of the IC 3E looking from the back of the printer. They can be seen in the bottom center of the diagram showing the DIP switch locations, and the ends to use are the ones closest to the DIP switches. With a bit of dexterity the leads can be soldered in place without removing the circuit board.

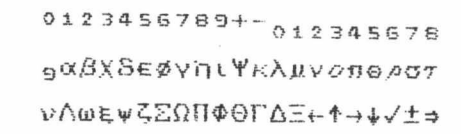
There are at least two versions of the printer with the original ROM. One of them has three 2716 EPROMs mounted in sockets 1B, 2B and 3B, and the jumper J1 alongside the 8049 has been cut (making the 8049 into an 8039). This gives 6K of ROM, 4K for the program and 2K for the character sets. The other version uses the 2K ROM in the 8049 and has 4K of external ROM in a single 2332 ROM mounted in socket 1B. The jumper J1 is left intact. The second version can easily be converted into an EPROM version, simply by cutting the jumper J1, and installing three 2716 EPROMs in the supplied sockets. The 8049 code is replaced by the code in socket 3B, the character set is in socket 2B, and the remaining program code is in socket 1B.

Incidentally, if anyone is interested in my alternative character set instead of the Japanese set I can provide a set of 3 EPROMs for $70, including air-mail postage. This will also allow ESC4 and

(Continued on page 74)

(Continued from page 72)

ESC5. A sample of the character set is given below.



Special requests for other characters can also be filled, but at a higher price. If there is sufficient demand for an APL or other standard set, they may be provided later at the above price.

Incidentally, I have written to Epson America some time ago asking whether sale of the modified ROMs was allowed, since presumably their program is copyrighted; they have not bothered to answer me. However, since the ROMs are of little use without an MX-80 printer, there should be no conflicts.

Sincerely
Arne Rohde
Pilevej 31, 7600 Struer
DENMARK

DDJ

BOOK REVIEWS

Ciarcia's Circuit Cellar Volume II
By Steve Ciarcia
Byte Books/McGraw-Hill
$12.95, 220 pages
Circle No. 459 on Reader Service Card
Reviewed by George W. Jolly

This is a collection of nineteen articles published in *Byte* magazine between December, 1978, and June, 1980. Fourteen of them were voted by *Byte* readers as the best article of the month. This collection receives a similar vote from me. Several times in the book letters from *Byte* readers are reprinted after the article to which they pertain. This adds an interactive touch to the book which is difficult to enjoy from the original magazine alone.

Interfacing microcomputers to the outside world is the main theme of this collection. A three-part series on Steve's home security system provides many ideas on the use of microcomputers for intruder detection. Flowcharts and block diagrams easily explain the design of such systems. Another article shows the use of the BSR X-10 home control system to automate lamps and appliances. One of the letters reprinted with this article describes an alternative method to that shown in the article.

Other articles deal with a variety of microcomputer topics. Steve explains how to use sound generation chips, joysticks, real-time clocks, and LED displays. Even a simple biofeedback monitor is described. Most articles are construction articles written in plain language and well-illustrated. Steve is very good about explaining why his circuits were designed the way they were. As a result you can learn a lot just reading the articles.

One interesting article describes Steve's home heating system. He has added a wood stove to his hot-water distribution system to supplement the oil burner. The article describes the problems involved in protecting the system from excessive heat, and how he used a microcomputer to automate the system. Circuits given in the article allow remote temperature sensing and electric motor control. Reading how Steve handled a practical design problem in the real world really helps in understanding the concepts.

Experimenters will find many circuits to play with in this collection. Fiber optic communication methods are explained, and the Intel 8086 microprocessor is described. Two articles show how to connect I/O devices to the TRS-80 Model I without the expansion interface. You may be interested in these articles even if you do not have the TRS-80 Model I because they provide examples of interfacing from which you can learn.

Ciarcia's Circuit Cellar Volume Two is a collection of excellent articles brought together in a convenient form. If you want to experiment with microcomputer interfacing, or want to read some entertaining design articles, give it a try.

Using Structured Design
By Wayne P. Stevens
John Wiley & Sons
$24.95, 231 pages
Circle No. 469 on Reader Service Card
Reviewed by Chuck Ballinger

This book, unlike others in the field of structured design, deals with the practical approach. When you finish this book you will have practical ideas and solutions to how you might make use of the structured techniques.

While written for the beginner to intermediate level programmer it is also full of information and ideas that encompass most of the available information on structured design. Managers, business people and long-time programmers stand to gain substantially from this book.

"Structured" is the concept of separating functions within a program or system into relatively independent modules.

This concept, while relatively new to the programming environment, has existed in industry for years. The component parts of a plane, a car, or a house and the necessary interface between their functional components is so well defined that they can be (and typically are) designed, manufactured, and improved by different companies or people. Moreover most parts are interchangeable among like designs. *Structured Design* teaches you to use comparable concepts in designing and writing your programs. Remember, the overall goal of structured *anything* is to reduce the cost of developing and maintaining that system or program. This book takes you through the realm of structured design admirably. The set of topics (structured design concepts, structure charts, coupling, binding, etc.) all relate to one another and do not create any large gaps that must be filled in later.

One of the strong points in this book is the section on Tips and Techniques. Most of us prefer the "show me" method whereby an idea or concept is explained and then demonstrated. This serves to enhance what you've just read as well as allowing you to test visually if this is what you had in mind. This book does this in an easy to understand fashion, while not being so simple that someone with some pre-existing exposure to structured design would not benefit from the text material.

In addition you are given the opportunity to improve an existing Structure Chart in chapter 8 which gives you an idea of what it is and then the steps you may want to take to improve your end product. You are also involved in generating an initial chart, since the hardest place to start is getting that first attempt down on paper. This approach should help you over that initial hurdle.

In short this book covers the many topics that make up the sometimes giant word "structured," and breaks them down to a level where you can feel comfortable about learning and using a really excellent technique. Even if you only write programs for yourself you should consider using this method. Your time is valuable also and structured design can lend itself to reducing your development time and thus freeing you for more projects that you just don't have time for now. This book is a very fine example of how to do it and should be read by all.

Computer Simulation in Business
By Hugh J. Watson
John Wiley & Sons
$22.95, 358 pages
Circle No. 479 on Reader Service Card
Reviewed by Robert Ashworth

Did you ever wonder how a business, firm or person might change over time? This book is for the college student or others who want to learn more about the art and science of simulation. It is based on the author's dissertation and 12 years of research in financial planning systems and human decision making. Simulation is defined as a mathematical model that describes the behavior of a system over time.

The text begins by identifying different models, either physical or symbolic. These in turn are divided into their purpose, mode of analysis, the way in which randomness is employed and the use of specific applications. A balanced approach is also evident as the author cites reasons when simulation is not useful.

Chapter II illustrates ten real busi-

ness applications covering such areas as finance, production, marketing, personnel and economics. A Monte Carlo or random simulation is used when variables like demand and delivery of a product are expressed as the frequency of occurrence. In another example a manual step-by-step method is applied to an inventory control problem.

One of the strengths of this book is its treatment of financial planning generations. The author shows how traditional line item budgeting can be incorporated into the desired simulation. Simple examples of both analytical and numeric methods are provided.

Probability distributions with their many variations are well developed. Supporting FORTRAN code is also helpful in specific cases. Several statistical tests are called upon as required to test assumptions about the real world.

The chapter on random numbers describes clearly what they are, how you can generate them, and what tests they must pass for acceptance. The midsquare and congruential methods are shown for both manual and computer applications.

The problem of waiting lines is well described; three pages of FORTRAN with comments describes the single channel, single phase case. The discussion is thoughtful and clearly shows how to handle the more difficult situations.

The next chapter deals with special-purpose simulation languages such as GPSS. Short discussions of DYNAMO, GASP, SIMSCRIPT and IFPS are helpful but could be longer.

The problem of human decision-making is illustrated with trust investment, vendor selection, personnel management and processing of overdrafts. Perhaps later editions will expand this difficult topic. Possibly a whole book would be helpful.

No book on simulation would be complete without a portion devoted to management games. Not only were several types of games presented, but the purpose, advantages, limitation and administration were candidly discussed. The chapter concludes with a detailed look at the Credit Union Management Game.

Corporate simulation models may be financial, marketing or production in structure. A telecommunications model is used for detail illustration. When it is related to the organization's planning process it becomes an effective tool to support the decision-making process.

The last two chapters deal with advanced techniques and procedures for the development of successful simulation models.

In summary, this is a very readable book on a rather technical subject. The chapters have a number of good problems and extensive references. To assist in understanding the many illustrations found in the text, the author has included an appendix of eight useful statistical tables. If you want to simulate part of the world, here's your opportunity. DDJ

DR. DOBB'S JOURNAL

For Users of Small Computer Systems

$2.50
($3.00 in Canada)

Number 70 August 1982

A "C" Command Line Processor
Anthony Skjellum

Performance Evaluation in Small Computers
Michael K. Kan

Intel's 8087: Numeric Data Processor
Ray Duncan

and more!

File Transfer Utilities . . .

. . . Elliptical Answers . . .

. . . IBM-PC Benchmarks

DR. DOBB'S JOURNAL
For Users of Small Computer Systems

August 1982 Volume 7, Issue 8

Editor – Marlin Ouverson
Assistant Editor – Fritzi Lareau
Contributing Editors –
Dave Caulkins, Dave Cortesi,
Ray Duncan, Michael Wiesenberg
Marketing Director – Craig Harper
Marketing Assistant – Beatrice Blatteis
Advertising Director – Laird Foshay
Circulation Manager – Gloria Romanoff
Circulation Assistants –
Theresa O'Rourke, Terri Marty
Art Director – Clifford West
Production Manager – Barbara Ruzgerian
Production Assistant – Margaret Tsai
Typesetter – Renny Wiggins
Cover Illustration – Margaret Tsai

© 1982 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $40 for first class to Canada and Mexico; $58 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Thomas H. Grohne, G. V. Elkins, Robert M. Connors, William J. McEown, Robert C. Luckey, Robert N. Stabler, DeWitt S. Brown, John B. Palmer, Burks A. Smith, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Howard E. Decker, Jim Wilson, Frank Lawyer, Chris Pettus, Paul Lutus, Ben Goldfarb, Rodney Black, Steven Weiss, Pat Phelan, John Brodie, James Fleming, Unison World, Thomas Davis, Jan Steinman, Ronald E. Johnson, G. Hunter. **Sustaining Subscribers:** Pete Roberts, Steven Fisher Enterprizes. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Hofacker-Verlag, Tegernseer Strass 18, D-8150 Holzkirchen, West Germany. **Asia & Australia:** ASCII Publishing Co., Ltd, Aoyama Building 5F, 5-16-1 Minami Aoyama, Minato-ku, Tokyo 107, Japan.

CONTENTS

Articles

10 Argum — A "C" Command Line Processor
by Anthony Skjellum
This program provides sophisticated command line parsing in order to help programmers produce more powerful and consistent code. It is portable, easily modifiable, and of interest to both 8- and 16-bit users.

34 Performance Evaluation in Small Computers
by Michael Kan
Performance evaluation is essential to any business; when it comes to squeezing the most from your micro, proper evaluation is critical.

41 SEND/RECEIVE File Transfer Utilities
by Lawrence Hughes
These utilities allow the user to transfer files from adjacent CP/M systems; source code is included.

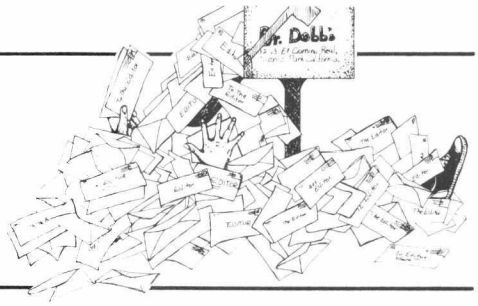
47 Intel's 8087 Numeric Data Processor
by Ray Duncan
A comprehensive look into the architecture and capabilities of the 8087 co-processor; or what to do with the empty space on your IBM-PC motherboard.

Departments

6 Letters
7 Editorial
8 Dr. Dobb's Clinic
51 Of Interest

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, Box E, 1263 El Camino Real, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025 · 415/323-3111

LETTERS



More C Notes

Dear Mr. Ouverson:

The May 1980 issue devoted to the language C was fantastic. The compiler written by Ron Cain was an excellent means of immersing oneself into the C language, along with the other articles. As Ron mentioned in closing, other people might have a problem getting the compiler operational. I was one of those people, and I thought my experience might be of interest to others.

First some background. I have access to a large CDC Cyber-172 but not any compiler for C, although a variety of other languages are available. Thus, the question of how to use the compiler as written quickly arose. It was as quickly realized that it would have to be rewritten into a different language. I chose to recode the program in Pascal, and get it operational. However, if the compiler was going to be running on the Cyber in Pascal, it would be useful to generate Cyber assembly language, Compass. I had considered leaving the code generation in 8080 mnemonics, yet the time delay in using a cross-compiler and then cross-assembler was not appealing. In implementing the compiler, some simplifications were possible. On current microcomputers, one has to be concerned with 8- and 16-bit items. Yet the Cyber-172 has 60-bit words, posing no problem for address items. At the same time, I chose to waste memory space, and only store one character in a word, wasting 54 bits per word. The rationale for this is to continue to interchange the role of integers and characters. At code generation, further simplifications were possible by using macros instead of calling runtime routines.

Recently, I completed teaching a course on Alternative Programming Languages, using APL, Ada and C. The experience in the classroom gave me and my students an opportunity to thoroughly test the compiler. As a result of student reactions, the compiler has been modified to handle the FLOAT data type. Again, with 60-bit words, FLOAT and DOUBLE can be considered the same.

In summary, the compiler has been recoded into Pascal, about 2200 lines, modified for floating point, and produces Cyber assembly language. Most of the runtime software is written in C, although the I/O, and error-handling routines are in Compass. The article by Chris Kern in *Byte*, May 1981, provided a version of PRINTF to use. My class developed our own version of SCANF.

Well, as Ron asked in the last part of the article, I thought he might like to know that a form of his compiler is alive and running well on a CDC Cyber-172.

Sincerely,
Freeman L. Moore
Computer Science Department
Central Michigan University
Mount Pleasant, MI 48859

Dear *Dr. Dobb's*,

I have adapted the small-c compiler by Ron Cain to generate code for the 6809. The adapted compiler is running on a South West Technical Products S/09 computer with 56K of RAM, running FLEX version 9.0. It generates assembly language acceptable to the SWTPC ASM09 assembler.

In addition to the features found on the original compiler, the following things have been added:

The type *unsigned*.
The operators "&&", "||", "?:", "~", "!".
The control structures "for", "do – while", "switch – case – default".
Octal and hexadecimal constants.

I also have a function library which implements the following:

- Command line arguments passed to main( ) through the argc, argv mechanism.
- I/O redirection using the "<" and ">" command line operators.
- The functions fopen, fclose, putc, getc, ungetc, getchar, putchar, iswhite, isdigit, isalpha, tolower, toupper, fgets, gets, fputs, puts, strlen, strcpy, strcmp, itoa, atoi, all of which match their definitions under UNIX. The I/O routines talk transparently to disk files or devices. They contain code for using ACIA's or PIA's, and they understand filenames like "/dev/tty", "/dev/modem", and "/dev/lp".
- The functions printf, fprintf, sprint, which are not quite as powerful as their UNIX counterparts.

The compiler does not prompt for source file names, nor compilation options but instead takes the source file names as command arguments and the options as UNIX-style command line flags. A sample command line might be:

cc -ps file1.c file2.c file.3c >xyz.asm

My main goal while making modifications was to make the system look as much like UNIX as possible. If all hobbyist implementations of C make themselves compatible with UNIX, then we will have a much easier time sharing software.

I can provide for $20 either all sources, executables, and documentation on a 5-¼-inch disk (not including the assembler, of course); or, listings of all source and documentation.

Anyone interested in forming a 6809 C user's group?

Dan Everhart
5823 202nd St. SW, Apt. #5
Lynnwood, WA 98036

He Wants the Whole Thing

Dear Marlin:

I just received my *DDJ* #68 this morning and read with great interest, the article on the multi-68000 system by Al Kossow. I urge you to please publish the listing in its entirety.

Yours truly,
E. E. Mullins
Box 73027
Houston, TX 77273

Is That You?

Dear *DDJ*:

We are interested in voice recognition hardware and software for an 8080 S-100 non-CP/M system for data entry.

Several years ago we saw ads for and articles about such items but can't find any now.

Can you list sources for them? Information from other readers will be appreciated.

Sincerely,
T. B. Finn
P. O. Box 2336
Goleta, CA 93118

Venus Rising

The Editor,

I am responding to Wayne Farmer's letter in *DDJ* #68, who lists a number of guidelines for a truly useful macro-assembler, and wonders whether one such exists. My own macro-assembler VENUS, which I have just completed, would seem to fit the bill.

VENUS is a structured macro-assembler, editor, symbolic debugger, linker, loader, and disassembler with a full complement of algebraic, string, and I/O facilities. It allows internal and external symbol names of eight characters,

and supports 16-, 8- and 3-bit externals (so that, for instance, the instruction

BIT FLAG, (IX+LOCN)

can be assembled, where both FLAG and LOCN are externals). It allows macros with string operators, can link up to 512 global symbols, will link programs of any size at all, and produces global cross-references among all modules. In addition to an IF-ELSE-ENDIF construction, it supports WHILE-ENDWH and DO-UNTIL control structures, and has a BREAK facility. Moreover, all of these control structures use the Z80 flags as condition tests, so that total control over register use is maintained.

Mr. Farmer also notes the problem of the awkwardness of needing to INCLUDE a large COMMON area along with each program module – thus increasing assembly time and requiring the reassembly of all segments when the COMMON is changed. VENUS has several facilities for handling this problem. First, it has an EXECUTE statement which allows a series of commands (get files, assemble them, link them, load them, list them, print them, etc.) to be executed without operator intervention. Thus if a program has 23 modules, a file may be created which, when EXECUTED, will leave the programmer free to do other things while the operations take place.

Second, the design philosophy of VENUS incorporates the notion that program development occurs normally in an accretionary fashion. Some parts of the program will be completed before others are developed. The completed modules can be separately linked and their symbol table of externals saved to disk. The modules under construction can be separately linked, and the linker knows the absolute locations of all declared ENTRY variables of the completed modules. Thus, if a program has 23 modules and 21 are completed, only the other two need to be reassembled and relinked during program development. When they are done, they may be added to the already completed programs, if desired.

The third VENUS facility for dealing with the problem of the COMMON area requires a bit of programmer discretion. VENUS allows the interleaving of normal code, data areas, common areas, and blank common areas, within program modules. Thus, there is normally no reason to INCLUDE a COMMON source file. However, like other assemblers, VENUS can perform only addition and subtraction on externals. If more complicated algebraic operations are required, they must be collected in one or more separate modules along with the EQUs that define the variables, and the results can be declared externals, to be drawn upon by other program modules. If the algebra needs to be changed, only the module involved need be reassembled before relinking the whole program.

(Continued on page 33)

Editorial

Thanks To Our Readers

Thanks go to the many readers who responded to our recent survey. Such questionnaires are some of the methods we use to get to know you a little better, to more precisely define what is looked for in *DDJ*. It is interesting to see how the tabulated results change from year to year as the industry presumably moves ahead and as different issues – telecommunications, software portability, new languages, etc. – ebb and flow in public opinion, if not in importance.

We were amazed, for instance, at how rapidly many of our readers have acquired IBM's Personal Computer. Equally interesting was that the 68000 chip has a greater following among our readers than the 8086/8088, although the margin is not overwhelming. Among topics eliciting very favorable response were CP/M, algorithms, assembly language, compilers and the Z80/8080. Pascal and Small-C/C were among the languages of evident enthusiasm, and multi-user operating systems are becoming worthy of more editorial attention.

This is a small and possibly predictable sample of the results we pulled from the survey. It helps to confirm what our mail, conversations and instincts tell us. As a token of our appreciation, three respondees were chosen at random to receive complete sets of *Dr. Dobb's Journal* bound volumes. They were T. E. Toews, Michael J. Young and Charles R. Rogers. The sets include the latest, volume number five, which was due to be released in mid-July.

* * *

For our newsstand buyers, a special note: Last year's Forth theme issue will be followed by another one next month. Since the first was nearly an immediate sellout, keep your eyes open for the September 1982 issue to appear. There are a number of features which are certain to be valuable not solely because of their use of Forth, but because of the solid design and programming technique they represent.

* * *

One last message for those of you who have been frustrated by the inaccuracy of your system clocks. If you aren't quite in step with solar time, it's because a leap second was added to our calendar on the last split-moment before midnight, June 30th. If it's not one thing, it's another . . .

– Marlin Ouverson
Editor

DR. DOBB'S CLINIC

by D.E. Cortesi

The Lazy Pinky and the H19

Our Heathkit H19 terminal has served us well for two years now, but recently we began to notice that, once in a while, when we typed the word "pascal" it came out on the screen as "pa[scal." Then stranger things began to happen. Once or twice a day we would type the word "photograph" or "graphics" – something with "aph" in it – and as the H key went down our full-screen editor would jump to the top of the file. Our editor has been patched to take the H19's F1 key as a signal to go to the top of the file.

Very strange. Why would "aph" look like "aph,F1"? With some experimentation we found that if we held down the A and P keys and pressed H, the computer would receive an "h" followed by the F1 escape sequence. And if we held down A and P and pressed S, the computer would get "s" followed by "[". More experiments revealed other combinations: holding down I and K and pressing L produced both "l" and "o".

We got out the schematic of the H19. The keyboard is encoded as a matrix of switches; the matrix is scanned and decoded by a single IC, a 5740. This chip sends pulses down the matrix columns; where a key is closed the pulse shows up on one of the matrix row lines. The chip converts the key address to a key number which is eventually decoded to ASCII.

When we mapped out all the four-letter combinations we had found (PA[S, OILK, etc.) on the keyboard matrix, it was quite noticeable that each combination was formed of keys that were at the corners of a rectangular subsection of the switch array. And then the light dawned. If any three key-switches at corners of a rectangle are closed, the key-switch at the fourth corner of the rectangle will also appear to be closed (see Figure 1 for an example). It is a classic example of what we used to call a "back-circuit" in the days when we repaired electrical accounting machines. This back-circuit is an inevitable feature of the H19's keyboard matrix; it could only be eliminated by putting a diode in series with every single key-switch.

So "pas[cal" was not a symptom of an intermittent failure in the H19; it had been there right along. How come it took 18 months to show up? That became clear after some thought. Your Intern is a touch-typist, and the P and A keys fall under his right and left pinky fingers, re-

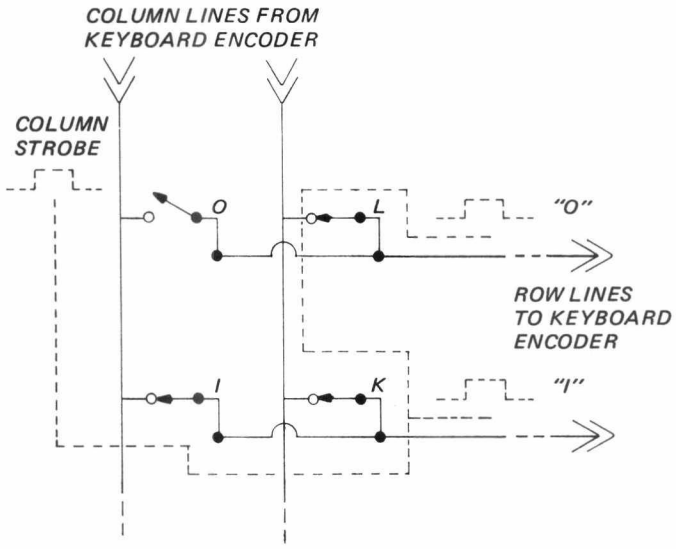


Figure 1.

Part of the key matrix of the H19 terminal. Holding I and K keys, pressing L key, produces spurious O-key output.

Listing 1.

```
100 REM Floating-point exercise
110 DEFDBL K,S
120 DEFINT I
130 DIM S(999,1)
140 PRINT CHR$(7) : REM signal to start the watch
150 FOR I=0 TO 999
160   K=K+.001
170   S(I,0)=SIN(K)
180   S(I,1)=COS(K)
190 NEXT I
200 PRINT CHR$(7) : REM stop the watch
```

Listing 2.

```
100 REM Sieve of Eratosthenes, after
101 REM Creative Computing, May, 1982
110 DEFINT A-Z : DIM X(5000)
120 PRINT "2"; CHR$(7); :REM start the watch
130 FOR P=3 TO 67 STEP 2
140    IF X(P) THEN 160 ELSE PRINT P;
150    FOR J=P*P TO 4999 STEP P+P : X(J)=1 : NEXT J
160 NEXT P
170 FOR P=71 TO 4999 STEP 2
180   IF X(P) THEN 190 ELSE PRINT P;
190 NEXT P
200 PRINT CHR$(7) : END
```

spectively. Clearly, our typing habits have grown sloppy over the past year. When typing a familiar word like "pascal," the pinky fingers are not coming off the keys fast enough. The P and A are staying down long enough so that, sometimes, when the ring-finger comes down on the S, there are a few milliseconds when all three keys are closed. The encoder chip then sees new closures of both the S and the [keys (or both the H and F1 keys, in "aph"). It's an interesting example of how a slow change in the habits of the *operator* can give the appearance of a developing bug in the *machine*.

Improvements SUBMITted

Robert Pasky of Newton Center, MA, has sent us fixes for the irritating problems that plague the SUBMIT command of CP/M. He worked out patches for both version 1.4 and 2.2 that cause SUBMIT to (a) retain lowercase letters from its input file, (b) pass all 32 control characters, and (c) accept empty input lines and pass them through to the output.

The last fix is particularly impressive, since we are told by Digital Research that that problem required too extensive a patch, and it would have to wait for a later version of the system.

We have combined all three of Pasky's patches with the one given in the May Clinic. Starting with a fresh, stock SUBMIT.COM file, use the L command of DDT to verify the instructions shown in Table 1, then change them as indicated. Save the resulting program as DDJ.COM. Then test it carefully before putting it into service.

Intern Benchmarks Machine

It seems to be all the rage these days to run benchmarks on the IBM Personal Computer. Since we've had a PC sitting next to our 4MHz Z80 system for some time now, we thought we'd join in the fun. Listings 2 and 3 show two programs that we ran on each machine. The Z80 system used Microsoft Disk BASIC version 5.0. Times were measured to the nearest second with a stopwatch, and averaged over several trials.

The floating-point exercise in Listing 1 ran in 32 seconds on the Z80 system; it took 40 seconds on the IBM PC. The program in Listing 2 is a good test of integer arithmetic and of array indexing. The Z80 did it in 18 seconds; the IBM in 24.

Tsk, tsk, tsk. The 8086 machine architecture is ever so much better suited to both these tasks than the Z80. It should be possible to write floating-point subroutines and array-indexing subroutines for the 8086 that would just run rings around the Z80. Judging by the CPU alone, IBM BASIC ought to be quite a bit faster; instead it is significantly slower. Why? The obvious guess is that the IBM interpreter was mechanically translated from 8080 to 8086 code, but we have seen absolutely nothing to verify that. It would be nice if some reader would disassemble the IBM ROMs and tell us if they look like translated code.

Elliptical Answers

Now we move from the mundane to the sublime, more or less. In a recent Clinic we published a call for help from Don Taylor of Corvallis, OR. He wanted, he said, a formula that would let him rotate the image of an ellipsoid (an elliptical solid) on all three axes. We received eight replies, each detailed, each full of (to us) daunting derivations. We panicked, packed up the lot of them, and sent them off to Don Taylor himself to analyze for us. What follows is his:

The Intern asked me to summarize the responses to the question that I posed in the April Clinic. What follows is, I hope, a better description of what I was trying to find, the solution that several readers found, and what it might be used for.

If you attempt to display three-dimensional objects on a monitor, you quickly discover the massive amount of calculation needed to model a complex object. Even relatively simple objects may need dozens of points for a rough representation. To attempt to manipulate one of these objects in real tim is no small task. Every point displayed is the result of at least one matrix multiplication, needing upwards of sixteen multiplies. Think of thirty or so points, updated five or ten times a second, and it becomes obvious why real-time, 3-d graphics are uncommon.

I began searching for another method of representing objects, one that would be flexible and yet give an order of magnitude increase in speed. It seemed that single points did not supply enough information about an object, given the amount of effort needed to find their apparent position. If we could model part of the object as a geometric solid, and find the perspective view of that whole subsection at once, some speed might be gained. The sphere seemed a likely candidate since its outline is easy to find and it is simple to draw. The only drawback is that the sphere just does not model many objects well. Then I hit upon the ellipsoid. It seemed the perfect solution; it can model a large class of objects with low resolution, needing only a few ellipsoids for any given object.

Two problems remained: how to find the perspective view of an arbitrary ellipsoid from an arbitrary point, and how to draw that view at high speed. The general feeling was that the view of an ellipsoid must be an ellipse, but there did not seem to be an easy way of proving that, nor of finding the formula that described the resulting ellipse.

A solution to the second problem, how to draw an arbitrary ellipse at high speed, was discovered by Pitteway (M. L.

(Continued on page 58)

Table 1.

--- Addresses ---		--- Instructions ---			
vers 1.4	vers 2.2	Verify		Change to	Purpose
0362	036F	ANI	5F	ANI FF	retain lowercase
0380	038D	MVI	M,00	MVI M,FF	allow empty lines
0434	0441	SUI	61	ANI 1F	allow both ^Z and ^z
043A	0447	MVI	A,19	MVI A,1F	allow ^[for ESCape
044C	0459	INR	A	NOP	" " " "
0516	0523	CPI	00	CPI FF	allow empty lines

Changes to SUBMIT. Using DDT, verify the instructions at the addresses shown, then change them as indicated. Save the changed file under a different name and test it.

Argum — A "C" Command Line Processor

by Anthony Skjellum

The Argum package is designed to simplify the process of command line argument interpretation. The package supports handling of dash options as well as command line arguments not part of a dash option sequence.

The job of processing command line sequences can be quite formidable if a reasonable number of dash arguments and other options are required of program operation. In general, each program has its own parser for handling these arguments. Not only does this lead to repetition of effort, but also spawns syntactical inconsistencies between programs. Such inconsistencies are often hard to remember and tend to frustrate users. In order to alleviate these problems, this package was developed.

Description

The Argum package is designed to handle dash options of the form

<-[|+] ><*s*>[<[:[|=]]*sub*>][<*arg*>]

where *s* is a single letter or a string beginning with a letter. (See more on string format below after **A_vset**.) As illustrated, the dash option may be invoked by a dash ('-') or, optionally, with a plus ('+') sign. Furthermore, the dash option may take two optional fields denoted the *sub-option* and *argument* fields. Considerable flexibility is afforded since each dash argument may use zero or more of these options. Furthermore, comand line arguments independent of dash options are made accessible separately.

Full error checking includes descriptive error messages and numerical return values which exactly describe the nature of the illegal specification.

A_ini: general initialization. The subroutine **A_ini** must be called to prepare the argument processing environment. **A_ini** takes several arguments which will be described in turn:

ARGUM base[ ];
unsigned dymax;

base must be an array of a size sufficiently large to handle the dash arguments possible for a given program. Each dash option requires one entry in the array. The unsigned quantity **dmax** is the number of elements in the array **base[ ]**.

char *nop[ ];
unsigned nmax;

nop is an array of pointers to strings. It is filled with pointers to arguments on the command line which are not associated with any dash option. The unsigned integer **nmax** is the size of the **nop** array, and hence is the maximum number of non-associated arguments permitted on the command line. If no non-associated arguments are expected, pass **nop** as **NULL** and **nmax** equal to zero. This will prohibit these arguments from being used.

nb: The user program may consult the global unsigned integer **A_ntyped** which is the number of arguments currently in the **nop[ ]** array. See Example #3.

A_set: initialize dash options. The subroutine **A_set** must be called for each dash option to be initialized. It returns a *dash descriptor* (type **DASH** which is a pointer to type **ARGUM**) for the dash argument initialized by the call. A description of **A_set**'s calling arguments follows:

char ch;

ch is the character for the dash option. Both upper and lower case letters are permitted and are considered *distinct*. Dash options using characters other than letters are prohibited (sequences of the form -<*digit*>? . . . ? are returned unchanged since they are presumed to be negative numbers).

unsigned opts;

The variable **opts** is the logical OR of one or more of the following option selectors:

A_NONE
This dash argument uses no options.

A_ARG
The dash option takes an argument.

A_SUB
The dash option takes a sub-option.

A_PLUS
Plus invocation permitted.

A_COLOP
Colon in sub-option is optional.

A_NOCOL
Colon not allowed to begin sub-option.

A_COLEQ
Colon character is an equal sign.

A program may not declare more than **dmax** dash options. **A_set** writes an error message and aborts execution if such an error occurs.

nb: **A_set** is used exclusively to define single character dash options. Use **A_vset** (see below) for setting dash options longer than one letter.

A_vset: Initialize variable length dash option. This subroutine operates similarly to **A_set** but with the following exceptions. While **A_set** takes a character as its first argument, **A_vset** takes a pointer to a null terminated string which is the dash option name. Furthermore, the options **A_COLOP** and **A_NOCOL** are not available.

char *str;

Pointer to the *matching string* containing the name of the dash option. No copy of the string is made.

This documentation and software package Copyright 1981 © Anthony Skjellum. All rights reserved. This software and documentation may be distributed for non-commercial uses only. Any commercial use, without the author's written permission, is prohibited.
Anthony Skjellum, 1695 Shenandoah Rd., San Marino, CA 91108.

Therefore, the user program should either use a constant string argument when calling A_vset or at least assure that the string contents will not change.

Matching Strings. When using multi-character dash options, it is often convenient to permit abbreviations. This is facilitated through the *matching strings* which contain the characters of the dash option name and optional *abbreviation delimiters* (the character '|'). These characters may occur after each character in the string except the last.

Matching strings are of the form

$$<char['|']><char['|']>...<char>$$

The *abbreviation delimiters* permit a long dash name to be specified by one or more abbreviations. Therefore, the string "mod|ule" defines a dash option name "module" which may be specified by the whole name or by "mod."

nb: The character '|' may not be part of a dash option name.

nb: During parsing, dash options are scanned in order of definition. When permitting abbreviations, the program should be careful to define dash options in such an order that ambiguity will not occur.

A_pars: command line processor. This subroutine is the general purpose command line processor which will be employed by most programs using this package. Its arguments are as follows:

int argc:
char *argv[ ];

The integer **argc** indicates the number of arguments in the array **argv[ ]** as per the C language command line convention.

int *i;

A_pars begins interpreting the command line starting at the element pointed to by **argv[*i]**. If ***i** is set to one, the whole command line is parsed. However, if *fixed* arguments are required by the program, it should first check for them and then call **A_pars** with the appropriate value for ***i**. Note that a *pointer* (as opposed to a *value*) is passed. This is done so that ***i** can be returned with the index of the offending argument when **A_pars** returns an error condition (a return value of less than zero).

nb: **A_pars** cannot be used with programs which expect dash option repetition. To produce command line parsing of this form, use **A_next** (described below).

A_argerr: parse and exit on error. This subroutine calls **A_pars** and then checks for an error condition returned from **A_pars**. If an error did occur, it calls the subroutine **A_err** described below. The result is a descriptive error message and an exit with a value of one.

Arguments same as **A_pars**.

A_next: process next argument. **A_next** contains the command line parser for the Argum package and is used by **A_pars** which does the complete parse. However, as noted above, **A_pars** is not designed to handle dash option repetition. When a program requires repetition, it must use **A_next** directly. Typically, a *do . . while* loop is used. After each execution of **A_next**, a check is made for the occurrence of a repetition. Appropriate action may then be taken by the program. The description of **A_next**'s arguments illustrate the mechanism by which repetition is handled. See also the second tutorial example.

int argc;
char *argv[ ];

These are the same as for **A_pars** and **A_argerr**.

int *i;

***i** is the element of **argv[ ]** to be parsed.

***i** is returned with its new value. The *do . while* loop should continue while ***i** is less than **argc**.

char *mode;
char flag;

These two variables are used to provide the capability to process option repetition. The **flag** variable must be OR'd with **A_REP** in order for option repetition to be permitted. When repetition occurs, the variable ***mode** is returned OR'd with **A_REP**. This is an indicator for the calling program to handle the repetition. Typically, the element of **base[ ]** will have to be moved to a safe place since a further repetition of this option will overwrite the contents of the specific element in this array.

The index of the array which was repeated is the return value of **A_next**.

nb: ***mode** is returned OR'd with **A_NOASC** if a non-associated argument was encountered.

nb: See also the function **A_copy** which provides a straightforward way to copy elements of the type **ARGUM**.

A_until: process until non-associated argument. This subroutine processes command line arguments until a non-dash argument is found. When repetition is permitted, it will return to the caller any time a repetition occurs. In case of repetition, **A_until** returns the index of the array **base[ ]** referenced.

int argc;
char *argv[ ];
int *i;
char *mode;
char flag;

Same as for **A_next**.

int *count;

Returns number of dash arguments processed.

A_while: process while non-associated argument. This subroutine is used to process a sequence of non-dash arguments. It stops when it encounters the first dash argument or when no more arguments are available. If this dash argument was a repetition, **A_while** returns the index of the array **base[ ]** referenced.

nb: The first dash option encountered by **A_while** is parsed.

int argc;
char *argv[ ];
int *i;
char *mode;
char flag;
int *count;

Same as for **A_until.**

A_err: Error handling. The subroutine **A_err** is called in order to display a message based on an error value generated by **A_pars**, **A_argerr**, or **A_next**.

int argc;
char *argv[ ];

As above.

int i;

Index of **argv[ ]** containing the offending argument. This is a *value*.

char error;

The **error** variable must contain a valid error number returned by one of the command line processors. Invalid error numbers result in the message *Bad syntax*.

char mode;

The **mode** variable instructs the subroutine on its further action after displaying the error message. If **mode** is zero, it returns to the calling subroutine. Otherwise it performs an **exit** function calling with parameter **mode.**

A_typed: tests is dash option typed. This function is used to determine if a dash argument was typed. For use after a successful command line parse.

DASH name;

The argument **name** is the *dash descriptor* for the dash option to be interrogated.

nb: **A_typed** and the functions **A_t_arg**, **A_t_sub** and **A_t_plus** (the latter three described next) are not subroutine calls but instead generate inline code for greater efficiency.

A_t_arg: tests if dash argument typed. This function tests for the presence of an argument associated with *dash descriptor* specified.

DASH name;

As above.

A_t_sub: tests if dash sub-option typed. This function tests for the presences of a sub-option associated with *dash descriptor* specified.

DASH name;

As above.

A_t_plus: tests for plus invocation. This function tests for the plus invocation of the dash argument associated with the *dash descriptor* specified.

DASH name;

As above.

A_copy: Copy element of type ARGUM. This utility is provided to aid in processing repetitious command line options. When a repetition occurs, this routine will be used to move the entry in the **base[ ]** array to a location setup by the program. **A_copy** provides the subroutine to do the actual operation of copying.

DASH dest;

The pointer to the destination.

DASH sourc;

The pointer to the source.

A_namexit: descriptive error message utility. This error utility is used by **A_err** to provide descriptive error messages. It prints the program name followed by the message.

char *str;

Null terminated string to be printed.

char *argv[ ];

As above. Used to extract the program name (addressed by **argv[0]**).

After displaying the error message, **A_namexit** interrogates the external variable **A_mode** for further action. **A_mode** is an integer variable defined by the Argum package for handling error exits. If the variable is nonzero, **A_namexit** will perform an exit function with argument **A_mode**. Otherwise it will return to the calling subroutine. The function **A_desexit** performs an identical check.

A_desexit: another descriptive message utility. This subroutine is designed specifically for displaying a specific command line element as well as an error message. The output is of the form:

program name: message: element

After printing, it performs its exit check as described for

A_namexit.

char *str;
Pointer to string containing the error message.

int indx;
Element of **argv[ ]** containing offending element.

char *argv[ ];
As above.

A_isarg: Check for dash option. This is a utility subroutine used by **A_next** to check for a dash option.

char *ele
ele should point to the argument to be tested.
char *chr;
char *chr; is returned with the dash character, if the test was successful.

char *pflag;
***pflag** is returned with a value of two if the dash option was invoked with a plus.

Notes On Use

Loading. The Argum processor requires the *include* file */usr/include/argum.h* which is specified by the following:

#include <argum.h>

It is also necessary to link the subroutine library for the package. This is done via a *-largum* at the loading stage.

Names used by Argum. The following is a list of names used by the Argum subroutine package. Use these names only in conjunction with this package.

ARGUM	**A_argerr**	**A_isalpha**	**A_set**
A_ARG	**A_argum**	**A_isarg**	**A_t_arg**
A_COLEQ	**A_base**	**A_match**	**A_t_plus**
A_COLOP	**A_coln**	**A_max**	**A_t_sub**
A_NOASC	**A_copy**	**A_mode**	**A_typed**
A_NOCOL	**A_desexit**	**A_namexit**	**A_until**
A_NONE	**A_digit**	**A_next**	**A_while**
A_PLUS	**A_err**	**A_nmax**	**DASH**
A_REP	**A_indx**	**A_nop**	
A_SUB	**A_ini**	**A_ntyped**	
A_act	**A_init**	**A_pars**	

Tutorial Examples

Example #1: Standard parse with A_argerr. Assume we have a program which requires an input file and two mutually exclusive dash options to specify program action on the input file. We require that the program read the standard input if the input file is omitted, and that is write to the standard output. Furthermore, in the case that the two mutually exclusive dash options are both typed, we want to abort execution.

The C example in Listing One fulfills the above specifications.

Listing 1

```
#include <stdio.h>          /* standard library */

        ... any other libraries ...

#include <argum.h>          /* Argum library */

main(argc,argv)
int argc;       /* argument count */
char *argv[];   /* pointer of array of pointers to arguments */
{
        /*
                declare any externals required by the program

        */

        /*
                declare any variables specific to the main
                program
        */

        FILE *input;

        /*
                this program requires a file desciptor
                for its input file
        */

        ARGUM base[3];      /* space for three dash options */

        /* now define dash descriptors */

        DASH i_fil;     /* input file dash descriptor */
        DASH f_dash;  /* descriptor for first of two command dashs */
        DASH g_dash;  /* descriptor for second command dash */


        /*
                we must also have an integer set to where
                parsing shall begin
        */

        int pos = 1;


                /* start at first argument */

/*
        This program uses no non-associated
        command line arguments so requires
        no nop[] array.
*/

A_ini(base,3,NULL,0);       /* call initializer */

/*
        Initialized for four dash options,
        no non-associated arguments
*/

i_fil = A_set('i',(A_ARG));
                /* input file -i takes an argument */

f_dash = A_set('f',(A_NONE));
                /* first command dash -f takes no arguments */

g_dash = A_set('g',(A_NONE));
                /* second command dash -g takes no arguments */

/*
        We have now initialized the argument processor
        and can now parse the command line
*/
```

```
A_argerr(argc,argv,&pos);  /* parse */
                /* A_argerr exits on error */

/*
        We have had a successful parse, now
        deal with command line typed:
*/

/*
        Check if neither -f nor -g was typed.
        If so, abort.
*/

if(!(A_typed(f_dash) | A_typed(g_dash)))
{
        fprintf(stderr,"\nneither -f nor -g specified\n");
        exit(1);
}

/*
        Check if both -f and -g were typed.

        If so, abort.
*/

if((A_typed(f_dash) + A_typed(g_dash)) > 1)
{
        fprintf(stderr,"\n-f,  -g are mutually exclusive\n");
        exit(1);

}


/*
        First check if an input file was
        specified.  If not, select standard input
        for reading.
*/



if(A_typed(i_fil))      /* -i */
{
        if(A_t_arg(i_fil)) /* name also specified */
                input = fopen(i_fil->a_addr,"r");
        else
        {
                fprintf(stderr,"\nno -i arg, assumed stdin\n");
                input = stdin;          /* standard input */
        }
}
else
        input = stdin;          /* select standard input */



/*
        Now act on -f or -g
*/

if(A_typed(f_dash))  /* -f typed */
{
        ... action for -f ...

        }
        else    /* -g typed */
        {
                ... action for -g option ...
        }

        ....
}
 /* end of program */
```

Example #2: Repetition processing with A_next. Assume that we need a program which is to perform operations on its input file in a given order. Such operations will be specified by dash options. Since a given operation could be specified more than once, repetition capability will be required.

The C example in Listing Two implements these specifications.

Listing 2

```
#include <stdio.h>          /* standard library */

        ...             /* any other required libraries */

#include <argum.h>          /* Argum library */

main(argc,argv)
int argc;       /* argument count */
char *argv[]; /* pointer to array of pointers to arguments */
{

        /*
                declare any externals required by the program
        */

        /*
                declare any variables specific to the main program
        */

        FILE *input;

        char mode;   /* mode variable for A_next use */

        int retn;       /* variable used in do...while loop */

        /*
                this program requires a file descriptor
                for its input file
        */

        ARGUM base[6];      /* space for six dash options */

        /*
                Make repitition areas for -a, -b
                which are the dash options which will be
                allowed to be used more than once.          */

        ARGUM a_rep[3],b_rep[4];

        char a_cnt = 0; /* repitition count for -a */
        char b_cnt = 0; /* repitition count for -b */

        /* now define dash descriptors */


DASH i_fil;     /* input file dash descriptor */
DASH a_dash; /* function a */
DASH b_dash; /* function b */
DASH c_dash; /* function c */
DASH d_dash; /* function d */
DASH e_dash; /* function e */


/*              we must also have an integer set to where
        parsing is to begin
*/

int pos = 1;
                /* start at first argument */

/*
        This program uses no non-associated arguments
        and therefore requires no nop[] array.
        Initialize for six dash options:
*/

A_ini(base,6,NULL,0);       /* initialize */

/*
        Now set the dash options individually:
*/

i_fil = A_vset("i|nput",(A_ARG));    /* form: -i fil or -input fil */

a_dash = A_set('a',(A_ARG));   /* function a arg */
```

```
b_dash = A_set('b',(A_ARG));   /* function b arg */

c_dash = A_set('c',(A_NONE));      /* function c */

d_dash = A_set('d',(A_NONE));      /* function d */

e_dash = A_set('e',(A_NONE));      /* function e */

/*
        We have now initialized everything and
        can now begin processing the command line.
*/
 do     /* begin loop */
 {

        retn = A_next(argc,argv,&pos,&mode,A_REP);
                        /* permit repitition */

        if(retn < 0) /* fatal error occurred */
                A_err(argc,argv,pos,retn,1);



                        /* indicate error and exit */

        /*
                Now check for repitition:
        */

        if(mode & A_REP) /* repitition occurred */
        {

                switch(base[retn].ch)  /*  act  based  on  dash  typed  */
                {

                        case 'a':
                                /* valid repitition */

                                if(a_cnt >= 3) /* overflow */
                                        repexit(argc,argv,pos);

                                A_copy(&a_rep[a_cnt++],&base[retn]);
                                        /* perform copy */

                                break;

                        case 'b':
                                /* valid repitition */

                                if(b_cnt >= 4) /* overflow */
                                        repexit(argc,argv,pos);

                                A_copy(&b_rep[b_cnt++],&base[retn]);
                                        /* perform copy */

                                break;
                        default: /* all others illegal */

                                fprintf(stderr,"\nIllegal repitition\n");
                                exit(1); /* leave */

                } /* end switch */

        } /* end if */

} while(pos < argc); /* continue while arguments */

/*
        We now have arguments.   Program would now act
        on them:
*/



if(!A_t_arg(i_fil)) /* file name not specified */
{

        fprintf(stderr,"\nNo input file name specified\n");
        exit(1);
}
else
{
        input = fopen(i_fil->a_addr,"r"); /* open file for input */

        if(input == NULL)
        {
                fprintf(stderr,"\nCannot open %s\n",i_fil->a_addr);
                exit(1);
        }
}

        /*
                Program would now handle arguments
                in a meaningful way:
        */
        ....

} /* end program */


/*
        Subroutine repexit
*/

repexit(argc,argv,i)
int argc;
char *argv[];
int i;
{

        fprintf(stderr,"\n%s: Repitition overflow at %s\n",argv[0],argv[i]);
        exit(1);

}
/* end subroutine */
```

Example #3: Using non-dash arguments. Assume that we have a program which requires two non-dash arguments as well as three dash arguments which direct program execution. The program takes its input from a file.

The C example in Listing Three demonstrates such a case.

Listing 3

```
#include <stdio.h>          /* standard library */

        ...             /* any other libraries */

#include <argum.h>          /* Argum library */

main(argc,argv)
int argc;
char *argv[];
{



        /*
                declare any externals required by the program
        */

        extern int A_ntyped;        /* number of non-associated args */

        /*
                declare any variables specific to the main program
        */


        FILE *input;   /* file descriptor for input */

        char mode;   /* mode variable for A_next use */

        int retn;       /* variable used in do...while loop */
```

```
    ARGUM base[3];      /* space for six dash options */

    char *nop[2]; /* make space for two non-associated arguments */

    /*
          we must also have an integer set to where
          parsing is to begin
    */

    int pos = 1;
                    /* start at first argument */

    /* now define dash descriptors */

    DASH i_fil;     /* input file dash descriptor */
    DASH f_dash; /* function f */
    DASH g_dash; /* function g */


    /*
          Initialize:
    */

    A_ini(base,3,nop,2);  /* call initializer */

    /*
          Now set dash options individually
    */

    i_fil = A_vset("i|nput",(A_ARG));     /* form: -i file or -input fil */

    f_dash = A_set('f',(A_NONE));         /* f function */

    g_dash = A_set('g',(A_NONE));         /* g function */

    /*
          We have now initialized can now
          call A_argerr
    */

    A_argerr(argc,argv,&pos);  /* parse; exit or error */

    /*
          Successful parse:
    */

    if(A_ntyped < 2) /* two non-associated args not typed */
    {
          fprintf(stderr,"\nTwo string args required\n");
          exit(1);
    }

    /*
          Program may now process input file specification
          and the elements of the *nop[] array
    */

    ....

} /* end of program */
```

Error message numbers and their meanings. These are the error numbers returned by **A_next**, **A_pars**, etc.

NUM	Meaning
-1	No arguments on command line (i.e. argc = 1)
-2	Illegal dash option
-3	Option repetition error
-4	No arguments available (i.e. starting position >= argc)
-5	Too many non-dash arguments
-20	Argument processing not initialized (no call to A_ini)
-21	Illegal plus invocation of dash argument
-22	Illegal sub-option use on a dash argument
-23	Colon character not allowed to start sub-option
-24	Colon required to begin sub-option
-25	Illegal character in matching string (i.e. '|')
-26	Null matching string

(Listing begins on page 24)

Argum Line Processor

(Text begins on page 10)

```
/*****************************************************************
*                                                               *
*       Argum.c                                                 *
*                                                               *
*       command line processing module.                         *
*                                                               *
*       created:        08/12/81        Unix 7 version          *
*       updated:        09/16/81                                *
*                                                               *
*       from the BDS C version          07/01/81                *
*                                                               *
*       by Anthony Skjellum                                     *
*       This documentation and software package Copyright 1981 ©*
*       Anthony Skjellum. All rights reserved. This software and*
*       documentation may be distrubuted for non-commercial     *
*       uses only. Any commercial use, without the author's written*
*       permission, is prohibited.                              *
*****************************************************************/

#include <stdio.h>
#include "_argum.h"


/*****************************************************************
*                                                               *
*               A_pars                                          *
*                                                               *
*       Expects:                                                *
*               argc number of arguments available in argv array.*
*               argv array of pointers to arguments (strings.)  *
*               i pointer to first element of argv at which     *
*               processing is to begin.                         *
*                                                               *
*       Returns:                                                *
*               A negative number indicating an error.          *
*               See A_err for information on error values       *
*               and their meanings.                             *
*                                                               *
*               Otherwise the processing was successful.        *
*                                                               *
*       The array "base" set by A_ini is configured according   *
*       to the arguments typed and various options utilized     *
*       or omitted.                                             *
*                                                               *
*****************************************************************/

A_pars(argc,argv,i)
int argc;       /* arg count of args stored in *argv[] */
char *argv[];   /* array of pointers to string arguments */
int *i;         /* place to start inside argv */
{

        /*      External variables used by argument processor */

        extern ARGUM *A_base;           /* ARGUM array to be used by
                                           program. */

        extern char **A_nop;            /* pointer to array of pointers to
                                           arguments (strings) which were not
                                           associated with any dash option */
```

```
A_next(argc,argv,i,mode,flags)
int argc;       /* arg count of args stored in *argv[] */
char *argv[];   /* array of pointers to string arguments */
int *i;         /* place to start inside argv */
char *mode;     /* used in option repitition processing */
char flags;     /* flags which control repitition etc. */
{

        /*      External variables used by argument processor */

        extern ARGUM *A_base;  /* ARGUM array to be used by
                                  program. */

        extern char **A_nop;    /* pointer to array of pointers to
                                   arguments (strings) which were not
                                   associated with any dash option */

        extern unsigned A_nmax;
                                /* maximum members of array pointed to by
                                   A_nop.  */

        extern unsigned A_ntyped;
                                /* actual number of members in *A_nop */

        extern char A_init; /* initialization flag */

        extern unsigned A_max; /* maximum number of dash options */
        extern unsigned A_act; /* actual # of dash args */

        /* variables used inside A_next */

        int j;          /* looping variable */
        char chr;       /* temp. char. variable */
        char *ptr;      /* temp. pointer to character var. */
        char found;     /* flag indicating valid option */

        int error;      /* error variable */

        /* begin code */

        if(!A_init) /* argument processing not initialized w/ A_ini */
                return(-20); /* flag the error */

        if(argc < 2) /* no args */
                return(-1); /* flag the condition */

        if(*i > (argc-1)) /* no args available past fixed arguments */
                return(-4); /* flag the error condition */

        *mode = '\0';       /* reset return value */

        {
                char *eptr;

                char pflag;

                ptr = argv[*i]; /* point to the ith argument */

                if(A_isarg(ptr,&chr,&pflag)) /* a dash argument */
                {
                        found = 0;

                        /* search for current dash letter in list
                           of legal dash arguments letters and strings */

                        for(j = 0;j < A_indx; j++) /* process */
                        {
                        if((error = A_match(j,ptr,&eptr,chr,,,, /* test *
                        {
```

```
        extern unsigned A_nmax;
                                /* maximum members of array pointed to by
                                   A_nop.  */

        extern unsigned A_ntyped;
                                /* actual number of members in *A_nop */

        extern char A_init; /* initialization flag */

        extern unsigned A_max; /* maximum number of dash options */
        extern unsigned A_act; /* actual # of dash args */

        /* variables used inside A_pars */

        char mode = 0;  /* repitition mode flag */
        char flags = 0; /* prohibit repitition */
        char error = 0; /* error flag */

        /* begin code */

        A_act = 0; /* # of args typed init. to zero */

        do {    /* loop until error or end */

                error = A_next(argc,argv,i,&mode,flags);

                if(error < 0) /* error has occurred */
                        return(error);

        } while((*i) < argc); /* end while */

        return(0); /* successful completion */

} /* end of A_pars */


/*************************************************************************
*                                                                        *
*               A_next                                                   *
*                                                                        *
*       A_next processes the next argument on the command line.          *
*                                                                        *
*       Normally, programs will use A_pars to handle the total           *
*       processing of the command line.  However, if command             *
*       line order is essential, or if dash arguments are allowed        *
*       to be repetitive, A_next must be used.                           *
*                                                                        *
*       A_next would then be called the appropriate number of times      *
*       with error checking and repitition processing handled in         *
*       the main routine.                                                *
*                                                                        *
*       Note that A_err can be used to signal standard errors and        *
*       then abort.                                                      *
*                                                                        *
*       Expects:                                                         *
*               argc, argv, as per A_pars.  The ith argument is          *
*       processed if possible.                                           *
*                                                                        *
*       if flags & A_REP, repitition is allowed.  If an option           *
*       repitition occurs,  *mode is returned OR'd with A_REP.           *
*                                                                        *
*       *mode is OR'd with A_NOASC if a non-associated arg was           *
*       encountered.                                                     *
*                                                                        *
*************************************************************************/
```

(Continued on top of page 24)

```
                if(error < 0) return(error); /* quit */

                ++A_act; /* # of dashes found */
                found = 1; /* flag it */
                break;

        } /* end if */

} /* end for */

if(!found)
        if(!A_digit(chr))
                return(-2); /* flag illegal syntax */

if(found)
{

        if(A_base[j].filled) /* option repitition */
        {
                {
                if(!(flags & A_REP))
                        return(-3); /* flag error */
                else
                        *mode |= A_REP;
                }
        }
        else
                A_base[j].filled = 1; /* set true */


        /* check for invokation with '+' */

        if(pflag == 2)
        {
                if(!A_base[j].plus)
                        return(-21); /* illegal '+' */

                A_base[j].fplus = 1; /* set it */

        } /* end if */


    {
    if((A_base[j].varln)||(!(A_base[j].colop |
                                A_base[j].nocol)))
    {
        {
        if((*eptr == A_coln)&&(*++eptr != '\0'))
        {
            /* we have a sub-option */

            {
            if(A_base[j].sub)
            {
                A_base[j].fsub = 1; /* flag it */
                A_base[j].s_addr = eptr;
                            /* point to it */

            }
            else return(-22); /* illegal sub */
            }

        }
        else
        {
            /* no sub-*/
```

(Continued on next page)

Argum Line Processor

(Listing continued, text begins on page 10)

```
            A_base[j].s_addr = NULL;
            A_base[j].fsub = 0;
        }

        } /* end enclosed if(sub-option) */

    } /* end if(A_base) */
    else
    {
        if((*eptr == A_coln)&&(A_base[j].nocol))
            return(-23); /* no colon allowed */

        if((!(A_base[j].colop))&&(*eptr != A_coln))
            return(-24); /* colon required */

        if(*eptr == A_coln) ++eptr;
                        /* look at next char */

        {
        if(*eptr != '\0') /* non-null sub */
        {
            if(!A_base[j].sub) /* sub illegal */
                return(-22); /* error */

            A_base[j].s_addr = eptr;
            A_base[j].fsub = 1;
        }
        else
        {
            A_base[j].s_addr = NULL;
            A_base[j].fsub = 0;
        }
        }

    } /* end if */

    } /* end enclosed if(A_base[j]....) */


    if(A_base[j].arg) /* takes an argument */
    {
        {
        if(*i < (argc -1 )) /* out of args ? */
        {
             ++(*i); /* increment i */

             ptr = argv[*i]; /* look at next */

             {
             if(A_isarg(ptr,&chr,&pflag))
             {
                 --(*i); /* back up */

                 A_base[j].a_addr = NULL;
                 A_base[j].farg = 0;

             }
             else
             {
                 A_base[j].farg = 1;
                 A_base[j].a_addr = ptr;
                        /* point */
```

```
A_until(argc,argv,i,mode,flags,count)
int argc;       /* arg count of args stored in *argv[] */
char *argv[];   /* array of pointers to string arguments */
int *i;         /* place to start inside argv[] */
char *mode;     /* used to flag non-associated/repitition conditions */
char flags;     /* flags which control repitition etc. */
int *count;     /* return value of number of arguments handled */
{

        /*      External variables used by argument processor */

        extern ARGUM *A_base;     /* ARGUM array to be used by
                                     program. */

        extern char **A_nop;      /* pointer to array of pointers to
                                     arguments (strings) which were not
                                     associated with any dash option */

        extern unsigned A_nmax;
                                  /* maximum members of array pointed to by
                                     A_nop.  */

        extern unsigned A_ntyped;
                                  /* actual number of members in *A_nop */

        extern char A_init; /* initialization flag */

        extern unsigned A_max; /* maximum number of dash options */
        extern unsigned A_act; /* actual # of dash args */

        /* variables used inside A_until */

        /* begin code */

        *count = 0; /* reset counting of args processed */

        do
        {
                int error;

                error = A_next(argc,argv,i,mode,flags); /* process next */

                if(error < 0) return(error); /* error exit */

                if(*mode & A_REP) /* repitition occurred */
                        return(error); /* index accessed */

                if(*mode & A_NOASC) /* non-associated argument typed */
                {
                        --(*i); /* point back at non-associated argument */
                        --A_ntyped; /* remove count of this argument */

                        return(0); /*  done processing */
                }

                ++(*count);

        } while(*i < argc); /* continue until no more args */

        return(0);

}
/*****************************************************************
*                                                                *
*       A_while                                                  *
*                                                                *
*       Process arguments until a non-associated argument        *
*                                                                *
*****************************************************************/
```

```
                        }
                        } /* end enclosed if */
                    }
                    else
                    {
                        /* ran out of args */

                        A_base[j].farg = 0;
                        A_base[j].a_addr = NULL;
                    }
                    } /* end enclosed if(*i < argc-1) */

                } /* end if (A_base[j].arg) */

                }
                else /* of if(found) */
                {
                        /* just an argument not attached
                           to a dash option */

                        if((!A_nmax)||(A_ntyped >= A_nmax)) /* overflow
                             return(-5); /* flag error */

                         A_nop[A_ntyped] = argv[*i]; /* save */
                         ++A_ntyped; /* increment count */

                         *mode |= A_NOASC;     /* unassociated */

                }
                        /* end if(found) */


        } /* end if (A_isarg) */
        else
        {
                /* argument w/out preceding dash arg. */

                if((!A_nmax)||(A_ntyped >= A_nmax)) /* overflow */
                        return(-5); /* abort */

                A_nop[A_ntyped] = argv[*i];
                ++A_ntyped;

                *mode |= A_NOASC;     /* unassociated */

        }
        /* end if */

    }

    ++(*i); /* increment *i to new value */

    return(j); /* index of A_base[] manipulated */



}

/*********************************************************************
*                                                                    *
*       A_until                                                      *
*                                                                    *
*       Process arguments until a non-associated argument            *
*                                                                    *
*********************************************************************/
```

(Continued on top of page 26)

```
A_while(argc,argv,i,mode,flags,count)
int argc;       /* arg count of args stored in *argv[] */
char *argv[];   /* array of pointers to string arguments */
int *i;         /* place to start inside argv[] */
char *mode;     /* used to flag non-associated/repitition conditions */
char flags;     /* flags which control repitition etc. */
int *count;     /* return value of number of arguments handled */
{

        /*      External variables used by argument processor */

        extern ARGUM *A_base;       /* ARGUM array to be used by
                                       program. */

        extern char **A_nop;        /* pointer to array of pointers to
                                       arguments (strings) which were not
                                       associated with any dash option */


        extern unsigned A_nmax;
                                    /* maximum members of array pointed to by
                                       A_nop.  */

        extern unsigned A_ntyped;
                                    /* actual number of members in *A_nop */

        extern char A_init; /* initialization flag */

        extern unsigned A_max; /* maximum number of dash options */
        extern unsigned A_act; /* actual # of dash args */

        /* variables used inside A_while */

        /* begin code */

        *count = 0;

        do
        {
                int error;

                error = A_next(argc,argv,i,mode,flags); /* process next */

                if(error < 0) return(error); /* error exit */

                if(*mode & A_REP) /* repitition occurred */
                        return(error); /* index accessed */

                if(!(*mode & A_NOASC)) /* a dash occurred */
                        return(error); /* exit */

                ++(*count);

        } while(*i < argc); /* continue until no more args */

        return(0);

}

/*********************************************************************
*                                                                    *
*       Same as A_pars, but prints a descriptive error message       *
*       (on error cond.) to stderr, then does an exit(1).            *
*       Assumes argv[0] points to program name.                      *
*                                                                    *
*********************************************************************/
```

(Continued on next page)

Argum Line Processor

(Listing continued, text begins on page 10)

```
A_argerr(argc,argv,i)
int argc;
char *argv[];
int *i;
{
        char error;

        error = A_pars(argc,argv,i); /* parse command line */

        if(error>=0) return(error); /* exit since no error */

        A_err(argv,argv,*i,error,1); /* write message and exit */
}

/*
        Output appropriate error message as set by A_args or A_argerr,
        then exit.
*/

A_err(argc,argv,i,error,mode)
int argc;
char *argv[];
int i;
char error;
char mode;
{
        extern char A_mode;

        A_mode = mode;  /* set exit status (zero is no-exit) */

        switch(error) {

                case -1:
                        A_namexit("No args",argv);
                        break;
                case -2:
                        A_desexit("Illegal dash option",i,argv);
                        break;
                case -3:
                        A_desexit("Option repitition",i,argv);
                        break;
                case -4:
                        A_namexit("No args available",argv);
                        break;
                case -5:
                        A_desexit("Too many non-dash arguments",i,argv);
                        break;
                case -20:
                        A_namexit("Arg processing not initialized",
                                        argv);
                        break;
                case -21:
                        A_desexit("Illegal plus invokation",i,argv);
                        break;
                case -22:
                        A_desexit("Illegal sub-option use",i,argv);
                        break;
                case -23:
                        A_desexit("Colon no allowed",i,argv);
                        break;
        if(dash == '-')
        {
                *chr = *++ele; /* character */

                if(isalpha(*chr))
                        return(1); /* valid arg */
        }

        return(0);
}

isalpha(chr)
char chr;
{
        chr = tolower(chr);

        if((chr >= 'a')&&(chr <= 'z')) return(1);

        return(0);
}

/*
        Initialization routine

*/

A_ini(base,dmax,nop,nmax)
ARGUM base[];
unsigned dmax; /* max # of elements to base */
char *nop[];   /* array of pointers to string arguments */
unsigned nmax; /* max # in nop */
{

        /*      External variables used by argument processor */

        extern ARGUM *A_base;     /*  ARGUM array to be used by
                                      program. */

        extern char **A_nop;       /* pointer to array of pointers to
                                      arguments (strings) which were not
                                      associated with any dash option */

        extern unsigned A_nmax;
                                    /* maximum members of array pointed to by
                                       A_nop.  */

        extern unsigned A_ntyped;
                                    /* actual number of members in *A_nop */

        extern char A_init; /* initialization flag */

        extern unsigned A_max; /* maximum number of dash options */
        extern unsigned A_act; /* actual # of dashes */
        extern unsigned A_indx; /* index for filling A_base */

        /* begin code */
        A_base = &base[0];        /* set A_base to base for future use */

        A_nmax = nmax;           /* max # of nop arguments */

        if(nop != NULL)
                A_nop = &nop[0]; /* point to first element */
        else
        {
                A_nop = NULL;
                A_nmax = 0;
        }
```

```
                case -24:
                        A_desexit("Colon required",i,argv);
                        break;
                case -25:
                        A_desexit("Illegal character error: ",i,argv);
                        break;
                case -26:
                        A_namexit("Program error: null matching string",
                                        argv);
                        break;

                default:
                        A_namexit("Bad syntax",argv);
                        break;

                } /* end switch */

}

A_namexit(str,argv)
char *str;
char **argv;
{
        extern char A_mode; /* error mode */

        fprintf(stderr,"\n%s: %s\n",argv[0],str);

        if(A_mode) exit(A_mode); /* exit if required */
}

A_desexit(str,indx,argv)
char *str;
int indx;
char **argv;
{
        extern char A_mode; /* error mode */

        fprintf(stderr,"\n%s: %s: %s\n",argv[0],str,argv[indx]);

        if(A_mode) exit(A_mode); /* exit if required */

}



/*
        A_isarg:

                checks if character addressed by ele is a
                dash option.
*/

A_isarg(ele,chr,pflag)
char *ele;
char *chr;
register char *pflag;
{
         char dash = 0;

        dash = *ele;
        *pflag = 1;

        if(dash == '+')
        {
                dash = '-';
                *pflag = 2;
        }
```

(Continued on top of page 28)

```
    A_indx = 0;             /* index of array being filled */
    A_max  = dmax;          /* max # of dash arguments */
    A_act  = 0;             /* reset actual # typed */
    A_ntyped = 0;           /* and reset # of args in A_nop */

    A_init = 1;             /* signify completed initialization */

    /* now for some error checking */

    if((!A_max)||((!A_nmax)&&(A_nop != NULL))) return(-1); /* illegal size */

    return(0); /* successful completion */
}

/*
    Single letter dash entry initializer
*/

DASH A_set(ch,opts)
char ch;
unsigned opts;
{
    extern ARGUM *A_base;
    extern unsigned A_indx;
    extern unsigned A_max;


    if(A_indx >= A_max)
    {
            fprintf(stderr,"\nA_set: too many dash options: -%c\n",ch);
            exit(1);
    }



    A_base[A_indx].ch = ch; /* set character */

    A_base[A_indx].filled = 0; /* not filled */
    A_base[A_indx].farg = 0;   /* arg not typed */
    A_base[A_indx].fsub = 0;   /* sub not typed */

    A_base[A_indx].arg  = !!(opts & A_ARG); /* expect an arg if true */
    A_base[A_indx].sub  = !!(opts & A_SUB); /* expect a  sub if true */
    A_base[A_indx].coleq =!!(opts & A_COLEQ); /* expect = not : if true */
    A_base[A_indx].plus = !!(opts & A_PLUS);
                                        /* allow plus invokation if true */

    A_base[A_indx].nocol = !!(opts & A_NOCOL); /* no colon permitted
                                                  to start sub-option */

    A_base[A_indx].colop = !!(opts & A_COLOP); /* colon not required */

    A_base[A_indx].varln = 0; /* not variable length */

    if(((opts & A_NOCOL) + (opts & A_COLOP)) > 1) /* mutually exclusive */
    {
            fprintf(stderr,"\nA_COLOP and A_NOCOL mutually exclusive\n");
            exit(1);
    }


    return(&A_base[A_indx++]); /* return pointer to entry */



}
```

(Continued on next page)

Argum Line Processor

(Listing continued, text begins on page 10)

```
/*
        Variable length dash entry initializer
*/

DASH A_vset(str,opts)
char *str;
unsigned opts;
{
        extern ARGUM *A_base;
        extern unsigned A_indx;
        extern unsigned A_max;


        if(A_indx >= A_max)
        {
                fprintf(stderr,"\nA_vset: too many dash options: -%s\n",str);
                exit(1);
        }

        A_base[A_indx].ch = *str; /* set character to first string char*/
        A_base[A_indx].strng = str; /* set pointer to string */
        A_base[A_indx].filled = 0; /* not filled */
        A_base[A_indx].farg = 0;    /* arg not typed */
        A_base[A_indx].fsub = 0;    /* sub not typed */

        A_base[A_indx].arg  = !!(opts & A_ARG); /* expect an arg if true */
        A_base[A_indx].sub  = !!(opts & A_SUB); /* expect a  sub if true */
        A_base[A_indx].coleq =!!(opts & A_COLEQ); /* expect = not : if true */
        A_base[A_indx].plus = !!(opts & A_PLUS);
                                        /* allow plus invokation if true */

        A_base[A_indx].colop = A_base[A_indx].nocol = 0;

        A_base[A_indx].varln = 1; /* variable length flag */

        if((opts & A_NOCOL)||(opts & A_COLOP))
        {
                fprintf(stderr,
                        "\nA_vset: illegal to specify conditional colon\n");

                exit(1);
        }


        return(&A_base[A_indx++]); /* return pointer to entry */

}

A_digit(chr)
char chr;
{
        if((chr<'0')||(chr>'9')) return(0);

        return(1);
}

/*
        Copies one element of type ARGUM into another
*/

A_copy(dest,sourc)
char *dest,*sourc;
                        if(*str == ':')
                                ++str;

                        continue; /* loop again */
                }

                return(0); /* failure */

        } /* end while */

        if(!i) return(-26); /* program error: null matching string */

        /*
                A successful search occurred
        */

        *eptr = ptr; /* point past matched string */

        return(i); /* positive return value indicates success */
    }
}
/*****************************************************************************
*                                                                            *
*       _argum.h                                                             *
*                                                                            *
*       header for command line processing module.                           *
*                                                                            *
*       created:        08/12/81        Unix 7 version                       *
*       updated:        09/04/81                                             *
*                                                                            *
*       from the BDS C version          07/01/81                             *
*****************************************************************************/

/*
        Constants
*/

#define A_NONE  0       /* readable way of writing zero options */
#define A_ARG   1       /* argument taken by dash option */
#define A_SUB   2       /* sub-option taken by dash option */
#define A_PLUS  4       /* plus option legal if used */
#define A_COLOP 8       /* colon in sub-option is optional */
#define A_NOCOL 16      /* colon not allowed to begin sub-option */
#define A_COLEQ 32      /* use alternate colon character, the equal sign */

#define A_REP   1       /* bit of flags variable set high if permitted */
#define A_NOASC 2       /* bit flags unassociated command line option */

/*
        Data type used for handling dash options
*/

typedef struct  A_argum {
        char    ch;             /* character of option */
        char    *strng;         /* pointer to string used for variable mode */
        unsigned filled : 1;    /* filled flag */
        unsigned farg   : 1;    /* filled argument */
        unsigned fsub   : 1;    /* filled sub-option */
        unsigned fplus  : 1;    /* used a plus instead of dash */
        unsigned arg    : 1;    /* flag set if argument possible */
        unsigned sub    : 1;    /* flag set if sub-option possible */
        unsigned colop  : 1;    /* flag set if sub-option colon optional */
        unsigned nocol  : 1;    /* flag set if colon prohibited */
        unsigned plus   : 1;    /* true if dash may be replaced by '+' */
        unsigned coleq  : 1;    /* true if alternate colon in use */
        unsigned varln  : 1;    /* variable length flag */
        char    *a_addr;        /* pointer to argument */
        char    *s_addr;        /* pointer to sub-option */
```

```
{
        int i;

        for(i=0;i<sizeof(ARGUM);i++)
                *dest++ = *sourc++;
}

/*****************************************************************************
*                                                                            *
*       A_match                                                              *
*                                                                            *
*****************************************************************************/

A_match(j,ptr,eptr,chr)
int j;                  /* subscript of A_base array */
char *ptr;              /* pointer to argument being processed */
char **eptr;            /* updated pointer to be returned to caller */
char chr;               /* character found by A_isarg */
{

    extern ARGUM *A_base;
    register int i; /* looping variable */
    register char *str;
    extern char A_coln;

    /* set colon character */

    if(A_base[j].coleq)
            A_coln = '=';
    else
            A_coln = ':';

    str = A_base[j].strng; /* pointer to string */

    if(!A_base[j].varln) /* standard char argument */
    {
            if(A_base[j].ch == chr)
            {
                    *eptr = ptr + 2; /* point past character field */
                    return(1);
            }
            else
                    return(0);
    }
    else /* process special string */
    {
            ++ptr; /* point past dash or plus */

            i = 0;

            while(*str != '\0')
            {
                    if(*str == ':') return(-25); /* illegal character err */

                    if((*ptr == '\0')||(*ptr == A_coln)) /* end of typed */
                    {
                            if(*--str == ':') /* valid stopping point */
                                break; /* break from while */

                            return(0); /* failed to match */
                    }

                    if(*str++ == *ptr++)
                    {
                            ++i;
```

(Continued on top of page 30)

```
                } ARGUM,*DASH;


/*      External variables used by argument processor */

        ARGUM *A_base;          /* pointer to ARGUM array to be used by
                                   program. */

        char **A_nop;           /* pointer to array of pointers to
                                   arguments (strings) which were not
                                   associated with any dash option */

        unsigned A_nmax ;
                                /* maximum members of array pointed to by
                                   A_nop.  */

        unsigned A_ntyped ;
                                /* actual number of members in *A_nop */

        char A_init ; /* initialization flag */

        unsigned A_max ; /* maximum number of dash options */
        unsigned A_act ; /* actual # of dash options */
        unsigned A_indx ; /* index used by A_set while filling
                                   in A_base */

        char A_mode ;     /* default to exit(1) condition */

        char A_coln ;   /* colon character for sub-option
                                   testing */

        /* end of externals */


#define A_typed(A)  A->filled
#define A_t_arg(A)  A->farg
#define A_t_sub(A)  A->fsub
#define A_t_plus(A) A->fplus


/*****************************************************************************
*                                                                            *
*       argum.h                                                              *
*                                                                            *
*       header for command line processing module.                           *
*                                                                            *
*       created:        08/12/81        Unix 7 version                       *
*       updated:        09/04/81                                             *
*                                                                            *
*       from the BDS C version          07/01/81                             *
*                                                                            *
*       by Anthony Skjellum                                                  *
*                                                                            *
*       Copyright 1981 (c) All rights reserved.                              *
*                                                                            *
*                                                                            *
*       For non-commercial uses only.  Any commercial use,                   *
*       without the author's written permission, is                          *
*       prohibited.                                                          *
*                                                                            *
*****************************************************************************/

#include "/usr/include/_argum.h"

        DASH A_set();
        DASH A_vset();
```

End Listing

Summary of Argum Definitions

Major Subroutines

A_ini: Initializer

```
A_ini(base,dmax,nop,nmax)
ARGUM base[ ]
unsigned dmax;
char *nop[ ];
unsigned nmax;
```

A_set: Dash option initializer

```
DASH A_set(ch,opts)
char ch;
unsigned opts;
```

A_vset: Alternate dash option initializer

```
DASH A_vset(str,opts)
char *str;
unsigned opts;
```

A_pars: Command line parser

```
A_pars(argc,argv,i)
int argc;
char *argv[ ];
int *i;
```

A_argerr: Parses with descriptive errors

```
A_argerr(argc,argv,i)
int argc;
char *argv[ ];
int *i;
```

A_next: process next argument

```
A_next(argc,argv,i,mode,flag)
int argc;
char *argv[ ];
int *i;
char *mode;
char flag;
```

A_until: process until non-associated argument

```
A_until(argc,argv,i,mode,flag,count)
int argc;
char *argv[ ];
int *i;
char *mode;
char flag;
int *count;
```

A_while: process while non-associated arguments

```
A_while(argc,argv,i,mode,flag,count)
int argc;
char *argv[ ];
int *i;
char *mode;
char flag;
int *count;
```

A_err: Report error message

```
A_err(argc,argv,i,error,mode)
int argc;
char *argv[ ];
int i;
char error;
char mode;
```

Utility Subroutines

A_typed: tests if dash name typed

```
A_typed(name)
DASH name;
```

A_t_arg: tests if dash argument typed

```
A_t_arg(name)
DASH name;
```

A_t_sub: tests if dash sub-option typed

```
A_t_sub(name)
DASH name;
```

A_t_plus: tests for plus usage

```
A_t_plus(name)
DASH name;
```

A_copy: copy structures of type ARGUM

```
A_copy(dest,source)
DASH dest;
DASH source;
```

Miscellaneous Subroutines

A_match: compare dash names to input

```
A_match(j,ptr,eptr,chr)
int j;
char *ptr;
char **eptr;
char chr;
```

A_namexit: error utility

```
A_namexit(str,argv)
char *str;
char (argv[ ];
```

A_desexit: error utility

```
A_desexit(str,indx,argv)
char *str;
int indx;
char *argv[ ];
```

A_isarg: checks for legal dash

```
A_isarg(ele,chr,pflag)
char *ele;
char *chr;
char *pflag;
```

A_digit: test for digit

```
A_digit(chr)
char chr;
```

A_isalpha: tests for letter

```
A_isalpha(chr)
char chr;
```

type definitions

```
typedef struct
        char        [
        char        ch;          /* dash character */
        char        *strng;      /* ptr for var. length mode */
        unsigned    filled : 1;  /* filled flag */
        unsigned    farg   : 1;  /* filled argument */
        unsigned    fsub   : 1;  /* filled sub-option */
        unsigned    fplus  : 1;  /* plus invocation */
        unsigned    arg    : 1;  /* set if argument possible */
        unsigned    sub    : 1;  /* set if sub-option possible */
        unsigned    colop  : 1;  /* set if sub-option colon optional */
        unsigned    nocol  : 1;  /* set if colon prohibited */
        unsigned    plus   : 1;  /* set if plus invocation okay */
        unsigned    coleq  : 1;  /* set if colon is replaced by equal */
        unsigned    varln  : 1;  /* variable length flag */

        char        *a_addr;     /* pointer to argument */
        char        *s_addr;     /* pointer to sub-option */

        ] ARGUM, *DASH;
```

Letters *(Continued from page 7)*

When I work out a distributor for VENUS, I will tell you!

Sincerely,
Herbert Gintis
Department of Economics
University of Massachusetts
Amherst, MA 01003

Dear *Dr. Dobb's*,

Douglas Crockford's suggestion (*DDJ* #66) for giving different names to distinct ADA dialects and look-alikes is basically sound, but a mere two dozen names probably won't suffice. The military alone will probably come up with more than that number of variants. Crockford suggests ARMADA and RADAR as being of particular military interest, but there are other probables on his list. (READABLE, admittedly, can be immediately discarded as totally inappropriate for a military environment.)

Among the two dozen are HEADACHE and GONADAL, likely candidates for the medical crowd. (It remains to be seen whether Walter Reed and Bethesda Naval will agree on the same dialect; my guess is that each will take one.) The Corps of Engineers will give ADAM thorough workout in flood control; CICADA will, of course, be reserved for the White House.

It is likely that more names will be needed, if only for the proliferation of military ADA dialects. I will suggest a few: The Department of Defense will begin with a single standard version of the language for code development. Name – DADA. (The Police, by the way, seem to be pulling for a subset, called DODO-DODA-DADA. Or was that DO-DO-DO-DA-DA-DA?) Interservice rivalry will cause rapid divergence from the standard, as the Army builds an ARMADA, the Air Force relies on RADAR and, I suppose, the Navy gets left with NADA. Then there is the regional specialization for different theaters of operation: ABADAN for the Persian Gulf and environs; ADAK for the North Pacific; CAPE-ADAR for the Antarctic; and so on... IBADAN, ENSENADA, LADAKH, etc.

Then there are dialects for special purposes: recreation and entertainment facilities will do their accounting in PINACOLADA; public relations computing jobs (below the Joint Chiefs level) will use ABRACADABRA....

Roy Smith
P.O. Box 4288
Berkeley, CA 94704

Editor's note: Roy's article went even further, but our prototype typesetting program (written in DELICADA) wouldn't print the clever but rather off-color finale. DDJ

Performance Evaluation
in Small Computers

by Michael Kan

Michael K. Kan, Scripps Memorial Hospital, 9888 Genesee Ave., La Jolla, CA 92038.

The evaluation of performance is essential in all branches of business or engineering. Through careful performance evaluation, one can find out the merits and deficits of a product and objectively compare various products. The evaluation of performance of a complete computer system is especially complex because it involves both hardware and software components, as well as the specific mixture of jobs to be run in the computer. A good understanding of computer performance evaluation is essential in choosing a cost-effective system to meet specific job requirements.

Many factors can affect the performance of a computer including the CPU speed, disk and printer speed, nature of algorithm of the application program, the computer language used, memory speed and size, the sophistication of the operating system being used and the work load. Computer performance evaluation is at present an important, emerging field that helps the system programmer to design, install and improve the operation of a computer system. I have chosen to illustrate some of the factors that affect the performance of a computer by program examples and precise time measurements in a small computer system.

Hardware Configuration

The computer used for the performance test programs is a Cromemco Z-80 microcomputer equipped with 32K bytes of high speed (250 nsec) RAM with no wait state necessary at machine cycles of 4 MHz. The CPU speed can be chosen between 2 MHz or 4 MHz by means of a two-way toggle switch on the CPU-board. This feature is extremely handy to demonstrate the effects of CPU speeds on computer performance in a given application. Other hardware includes a console and floppy disk interface (Cromemco 4FDC board) and two serial ports (Cromemco Tuart board). The disk drive is a Persi 277 drive for 8-inch single-sided, single-density diskettes.

One of the two serial ports (port B) on the Tuart board is used as the printer output port. This is normally operated at 300 baud for a Texas Instruments Silent 700 thermal printer. For comparing the effects of changing the printer speed on the program performance, the baud rate of the printer output port is changed by software to 9600 baud and a CRT is attached to the printer output port to simulate the presence of a fast printer. This port is operated at 9600 baud. Although the performance evaluation described in this paper was carried out in a Cromemco computer, the same program can be adapted to run on other computer systems to study the effects of CPU, I/O channels, computer languages and algorithms on computer performance.

A system clock is necessary to provide accurate time measurements of the system elapse times and printer I/O times in different version of test programs. Since a system clock is not available on the Cromemco computer, an external clock has been added. Communication between the system clock and the Cromemco computer is carried out by a second serial I/O port on the Tuart board. The construction and operation of the external system clock was described in an earlier paper published in the May 1982 issue of *Dr. Dobb's Journal*[1].

Test Programs

Two different programs have been designed to test the performance of the computer using different CPU speed, printer I/O speed and different languages. The first test program is a random number generator based on a linear mixed congruential algorithm. This represents an I/O-bound job, since the calculation of the random number is relatively simple, involving only a few multiplications, additions and divisions for each random number.

The second test program is a prime number generator. It has two versions. The first version (Prime1) attempts to find the first n prime number by a brute force approach. It simply examines a number of magnitude n and checks to see if there is any number between 2 and $n-1$ that divides n without any remainder. n is a prime number if it is not divisible by any number. This is a very slow algorithm because as n becomes larger, the number of divisions becomes proportionally large. This kind of prime number generator serves as an example of a CPU-bound job.

The second generator (Prime2) illustrates the great importance of an efficient algorithm to achieve good performance. Although the Prime2 program only incorporates three changes as described in the FORTRAN program (Listing 6), it improves the computational speed tremendously. A good discussion of prime number generator algorithms can be found in Niklaus Wirth's text[2].

All test programs are implemented in both BASIC and FORTRAN for the comparison of performance of the two languages. Cromemco 16K Extended BASIC v. 5.4, which requires a minimum of 32K bytes to run was the version used. The FORTRAN program is compiled by an early version of the Cromemco FORTRAN IV that will run in a 32K system (the later version will require 64K bytes of memory and support more functions). The various BASIC and FORTRAN programs are in Listings 1-6.

Test Runs

All test programs are run under the same hardware configuration within a period of two days to minimize any environmental changes. The output formats of all programs are similar (Figures 1-2). The BASIC programs are first tested for correctness and then stored on disk. Similarly, the FORTRANs are compiled and tested before storing as relocatable modules on disk. The FORTRAN system routines are linked to the relocatable modules by the linker each time the test run is performed.

Apart from the test programs, two other programs are used to keep track of the system elapse time and to print out the clock times at the end of each test run. These two programs and the test program are stored in a command file for batch operation. The first program in the command file is a "Jobbeg" program that sends commands to the clock simulator to reset all clocks and turn on the system clock just before loading the test program specified in the command file. Each test program has incorporated instructions to turn on the printer clock just before a random or prime number is printed on the printer, and turn off the printer clock after completion of printing. The last program in the command file is a "Jobend" program which sends a command to the clock simulator to stop the system clock and then gets the system and printer

clock data and prints them out on the console. The Jobbeg and Jobend programs were described in the previous paper[1].

Results

The total elapse time and printer I/O time for each test run are given in Tables 1-3. The random number generator programs illustrate the effects of I/O channel speed on the elapse time. For example, the elapse time averages 5 to 6 minutes for output at 300 baud on a slow printer. The elapse time is reduced to 1 to 2 minutes for faster output at 9600 baud. The speed improvements will be more prominent if a longer run is performed by generating, for example, 10,000 random numbers instead of 500 random numbers. This is true because the system overhead time used in loading the program will then be relatively small compared to the total elapse time. Nevertheless, the random number generator test runs do illustrate the characteristics of an I/O-bound job. The elapse time in such cases will not be affected greatly by the CPU speed or by the language used. The use of a high-speed printer, however, will dramatically reduce the system elapse time because the printing time is the bottleneck of the whole system.

The Prime number generator test runs are extremely interesting because they serve to illustrate several important factors in computer performance. It can be seen that the most dramatic improvements in the speed of our test programs can be achieved by a good algorithm. For example, the Prime2 programs outperform the Prime1 programs by a factor up to 37.2:1. The various elapse time ratios between Prime1 and Prime2 programs are summarized in Table 4. If we only consider the Prime1 programs, and assuming that a faster algorithm were not available, then the CPU speed will govern the elapse time. The Prime1 program is an example of a CPU-bound program. By increasing the CPU speed from 2 MHz to 4MHz, we can achieve a 2:1 improvement in elapse time. It is also noted that in the Prime1 program, the FORTRAN program outperformed the corresponding BASIC programs by a factor of 5-6:1. It is generally true that a compiled program will run much faster than an interpreted program. The performance comparison between FORTRAN and BASIC for the Prime1 program is summarized in Table 5.

Discussion

The use of performance analysis in a computer system is very helpful in identifying poor performance due to bottlenecks. For example, if the job of a computer system is to generate random numbers continually on a printer, and one would like to have as many random numbers in the shortest time, the most effective way of doing so is to remove the bottleneck by providing a fast printer. Of course, the above example is only a hypothetical example and in a real computer system the bottleneck may be due to slow disk access time or due to insufficient main memory. There are

Table 1.
Elapse and Printer I/O Times for Generating 500 Pseudo-Random Numbers

Baud Rate	CPU at 2 MHz Elapse Time	CPU at 2 MHz Printer Time	CPU at 4 MHz Elapse Time	CPU at 4 MHz Printer Time
BASIC				
300 Baud	5 min. 43 sec.	4 min. 54 sec.	5 min. 42 sec.	4 min. 42 sec.
9600 Baud	1 min. 7 sec.	12 sec.	1 min. 1 sec.	12 sec.
FORTRAN				
300 Baud	6 min. 3 sec.	4 min. 45 sec.	5 min. 48 sec.	4 min. 48 sec.
9600 Baud	1 min. 40 sec.	20 sec.	1 min. 13 sec.	11 sec.

Table 2.
Elapse and Printer I/O Times for Generating 500 Prime Numbers in BASIC

Baud Rate	CPU at 2 MHz Elapse Time	CPU at 2 MHz Printer Time	CPU at 4 MHz Elapse Time	CPU at 4 MHz Printer Time
Prime1				
300 Baud	3 hr. 1 min. 40 sec.	4 min. 42 sec.	1 hr. 33 min. 22 sec.	4 min. 37 sec.
9600 Baud	2 hr. 57 min. 22 sec.	49 sec.	1 hr. 28 min. 56 sec.	13 sec.
Prime2				
300 Baud	9 min. 3 sec.	4 min. 42 sec.	7 min. 2 sec.	4 min. 37 sec.
9600 Baud	4 min. 46 sec.	26 sec.	2 min. 36 sec.	13 sec.

Table 3.
Elapse and Printer I/O Times for Generating 500 Prime No.'s in FORTRAN

Baud Rate	CPU at 2 MHz Elapse Time	CPU at 2 MHz Printer Time	CPU at 4 MHz Elapse Time	CPU at 4 MHz Printer Time
Prime1				
300 Baud	34 min. 0 sec.	4 min. 31 sec.	19 min. 31 sec.	4 min. 23 sec.
9600 Baud	29 min. 52 sec.	28 sec.	15 min. 20 sec.	15 sec.
Prime 2				
300 Baud	6 min. 1 sec.	4 min. 34 sec.	5 min. 49 sec.	4 min. 39 sec.
9600 Baud	1 min. 53 sec.	28 sec.	1 min. 22 sec.	15 sec.

Table 4.

Elapse Time Ratios Between Versions of Prime Number Generators

Baud Rate	CPU at 2 MHz Elapse Time	Ratio	CPU at 4 MHz Elapse Time	Ratio
BASIC				
300 Prime1	181 min. 40 sec.	20.07	93 min. 22 sec.	13.27
Prime2	9 min. 3 sec.		7 min. 2 sec.	
9600 Prime1	177 min. 22 sec.	37.20	88 min. 56 sec.	34.20
Prime2	4 min. 46 sec.		2 min. 36 sec.	
FORTRAN				
300 Prime1	34 min. 0 sec.	5.65	19 min. 31 sec.	3.35
Prime2	6 min. 1 sec		5 min. 49 sec.	
9600 Prime1	29 min. 52 sec.	15.85	15 min. 20 sec.	11.21
Prime2	1 min. 53 sec.		1 min. 22 sec.	

Table 5.

Elapse Time Ratios Between BASIC versus FORTRAN Prime1 Program

Baud Rate	CPU at 2 MHz Elapse Time	Ratio	CPU at 4 MHz Elapse Time	Ratio
300				
BASIC	181 min. 40 sec.	5.34	93 min. 22 sec.	4.78
FORTRAN	34 min. 0 sec.		19 min. 31 sec.	
9600				
BASIC	177 min. 29 sec.	6.00	88 min. 56 sec.	5.83
FORTRAN	29 min. 52 sec.		15 min. 20 sec.	

methods available that help determine the bottleneck of a computer system so that appropriate measures can be taken in a cost-effective manner. A good discussion of computer performance analysis can be found in Ferrari's text[3].

I have also tried to emphasize in the test programs the importance of a good algorithm for efficiency and economy of computer utilization. The prime number generator is a good example of the large degree of speed improvements one can achieve without any modification to the hardware by using a good algorithm. As shown in our testing runs, a better algorithm for prime numbers (Prime2) can reduce the total elapse time by a factor of over 37:1. Many other important examples can be found in commercial and scientific applications. The use of a good algorithm is extremely important in sorting and searching. Comprehensive discussion of various sorting & searching algorithms can be found in Knuth's[4] and Lorin's text[5]. Other examples of good algorithms can be found in applications of graphs, path-finding problems, matrix multiplication, fast fourier transformation and pattern-matching algorithms[6].

References

1. Kan, Michael: "Adding a System Clock." *Dr. Dobb's Journal* No. 67, pp. 30-38, May 1982.
2. Wirth, Niklaus: *Systematic Programming: An Introduction*. pp. 137-141, Prentice-Hall, 1973.
3. Ferrari, D.: *Computer System Performance Evaluation*. Prentice-Hall, 1978.
4. Knuth, D. E.: *The Art of Computer Programming*, Volume 3. "Sorting and Searching." Addison-Wesley, Reading, 1973.
5. Lorin, Harold: *Sorting and Sort Systems*. Addison-Wesley, Reading, 1975.
6. Aho, A.; Hopcroft, J.E.; Ullman, J.D.; *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, 1974.

DDJ

(Listing begins on page 38)

Figure 1.

Console and printer output from pseudo-random number generator

```
RUN
          BASIC PSEUDO-RANDOM NO. GENERATOR

HOW MANY RANDOM NUMBER TO GENERATE:  500
DIRECT OUTPUT TO PRINTER

PSDUDO-RANDOM NUMBERS:


.063053         .017092         .104892
.050776         .093625         .739222
.396512         .135222         .179806
.343533         .695523         .855652
.698364         .457697         .806866
.172751         .030697         .349075
.680838         .920559         .315103
.691544         .604361         .297998
.044798         .378459         .251639
.148653         .045417         .153369
.263255         .137400         .201680
.268237         .835587         .527572
.641538         .505804         .032254
.812054         .719176         .040344
.389199         .127783         .587211
.833924         .558453
```

Figure 2.

Printer output from prime number generators

```
PRIME NUMBERS:

  1          41          101
  2          43          103
  3          47          107
  5          53          109
  7          59          113
 11          61          127
 13          67          131
 17          71          137
 19          73          139
 23          79          149
 29          83          151
 31          89
 37          97
```

Performance Evaluation

(Text begins on page 34)

Listing 1.

Pseudo-random number generator in BASIC

```
   10 REM
   20 REM
   30 REM BASIC PSEUDORANDOM SEQUENCE GENERATOR
   40 REM N = NO. OF RANDOM NUMBER TO BE GENERATED
   50 INTEGER N
   60 PRINT"           BASIC PSEUDO-RANDOM NO. GENERATOR"
   70 PRINT
   80 INPUT"HOW MANY RANDOM NUMBER TO GENERATE:  ",N
   90 REM
  100 REM INITIALIZE MULTIPLIER(M1), MODULUS(M2), INCREMENT(K)
  110 REM           SEED(S)
  120 M1=12923.0
  130 K=6949.0
  140 M2=31767.0
  150 S=2003.0
  160 PRINT"DIRECT OUTPUT TO PRINTER"
  170 OPEN\2,W\"$LP"
  180 PRINT\2\"PSDUDO-RANDOM NUMBERS:"
  190 PRINT
  200  FOR I=1 TO N
  210  R=S/M2
  220  S=FRA((M1*S+K)/M2)*M2
  230  REM FRA IS THE FUNCTION THAT RETURNS ONLY THE FRACTIONAL PORTION
  240  REM AFTER A DIVISION. S IS THE NEW PSEUDO-RANDOM NO. BETWEEN
  250  REM 1.0 - 32767.0
  260  REM STARTING PRINTER CLOCK BY SENDING COMMAND 3 TO CLOCK I/O PORT
  270  OUT%00F1%,%0033%
  280  PRINT\2\USING"        .######",R
  290  REM STOP PRINTER CLOCK BY SENDING OUT COMMAND 4 TO CLOCK I/O PORT
  300  OUT%00F1%,%0034%
  310  NEXT I
  320 PRINT"NO. OF RANDOM NUMBERS GENERATED = ",N
  330 CLOSE\2\
  340 BYE

>>
```

Listing 2.

Pseudo-random number generator in FORTRAN

```
C
C RANDOM.FOR MK/10-30-80 PSEUDORANDOM SEQUENCE GENERATOR
C
C
        PROGRAM RANDOM
C
        REAL MULTIP,INCREM,MODULU,INTRAN,SEED
        REAL REALRN
        BYTE TIMER,BEGCLK,STOPCK
        DATA TIMER,BEGCLK,STOPCK/Z'F1',Z'33',Z'34'/
C
C MULTIP=MULTIPLIER, INCREM=INCREMENT, MODULU=MODULUS
C REALRN=REAL RANDOM NO. BETWEEN 0 - 1.0
C
        WRITE(5,10)
10      FORMAT(1X,'          PSEUDORANDOM NO. GENERATOR')
C
C INITIALIZATION
        WRITE(5,11)
11      FORMAT (/,1X,'HOW MANY RANDOM NUMBER TO GENERATE  ')
        READ(5,12) MAX
12      FORMAT (I6)
```

```
  340 PRINT"NO. OF PRIME NUMBERS GENERATED  ",I
  345 PRINT
  346 PRINT
  350 CLOSE\2\
  360 BYE

>>
```

Listing 4.

First prime number generator in FORTRAN (Prime1)

```
C MK/10-28-80
C FORTRAN PRIME NUMBER GENERATOR
C
        PROGRAM PRIME
        INTEGER TESTER
        INTEGER MAX, PRINUM, CKLIMT
        BYTE TIMER,BEGCLK,STOPCK
        DATA TIMER,BEGCLK,STOPCK/Z'F1',Z'33',Z'34'/
C
C TIMER = I/O PORT ADDRESS (HEX F1) FOR CLOCK SIMULATOR
C BEGCLK = COMMAND TO START PRINTER CLOCK (HEX 33 OR ASCII 3)
C STOPCK = COMMAND TO STOP  PRINTER CLOCK (HEX 34 OR ASCII 4)
C
C
C MAX = MAXIMUM NO. OF PRIME NUMBER TO BE CALCULATED
C PRINUM = PRIME NUMBER
C CKLIMIT= LIMIT FOR CHECKING PRIME NUMBER.
C THE CURRENT PRIME1.FOR PROGRAM THE CKLIMIT IS SET TO BE NUMBER-1.
C
C
        WRITE(5,100)
100     FORMAT (10X,'FORTRAN PRIME NUMBER GENERATOR',/)
        WRITE(5,110)
110     FORMAT(1X,'ENTER NUMBER OF PRIME NUMBERS DESIRED:  ')
        READ (5,120) MAX
120     FORMAT (I6)
        WRITE(5,125)
125     FORMAT(/,1X,'DIRECT PRIME NUMBER OUTPUT TO PRINTER',/)
        WRITE(2,130)
130     FORMAT(/,1X,'PRIME NUMBERS: ',/)
        PRINUM=1
        WRITE(2,140) PRINUM
140     FORMAT (5I15)
        PRINUM=2
        WRITE(2,140) PRINUM
C PRIME1.FOR USES A SLOW ALGORITHM TO GENERATE PRIME NUMBERS
C
C FOR GENERATION OF PRIME NUMBER FROM 3 TO MAXIMUM NO. (MAX),
C ALL ODD NUMBERS ARE TESTED TO SEE IF THEY ARE DIVISIBLE BY
C ANY NUMBER BETWEEN 2 AND CKLIMT, WHERE CKLIMT IS EQUAL TO
C THE MAGNITUDE OF THE NUMBER BEING CONSIDERED - 1.  THE NUMBER IS
C CONSIDERED AS PRIME NUMBER ONLY IF IT IS NOT DIVISIBLE BY ANY NUMBER
C BETWEEN 2 AND CKLIMT.
C
C IF THE NUMBER (N) IS EXACTLY DIVISIBLE BY ANY NUMBER BETWEEN 2 TO
C CKLIMT, IT IS DISCARDED AS BEING PRIME NUMBER BY EXITING THE LOOP
C BY THE GOTO 300 STATEMENT
C
C
C LET NUMPRI=NO. OF PRIME NUMBERS GENERATED SO FAR
C
        NUMPRI=2
        DO 300 N=3,32767
        CKLIMT=N-1
        DO 200 I=2,CKLIMT
        TESTER=(N/I)*I
        IF (N.EQ.TESTER) GOTO 300
200     CONTINUE
C
C NUMBER NOT DIVISIBLE BY ANY NUMBER BETWEEN 2 TO CKLIMT, IT IS A PRIME
C NUMBER
C
```

```
C
        WRITE(5,15)
15      FORMAT (1X,'DIRECT OUTPUT TO PRINTER',/)
        MULTIP=12923
        INCREM=6949
        MODULU=31767
        SEED=2003
        WRITE(2,20)
20      FORMAT(1X,' THE FIRST 100 RANDOM NO. BETWEEN 0 - 1.0 :',/)
        DO 100 I=1,MAX
        REALRN=SEED/MODULU
        SEED=AMOD((MULTIP*SEED+INCREM),MODULU)
C
C TURN ON PRINTER CLOCK BY SENDING COMMAND 3 TO CLOCK I/O PORT
C
        CALL OUT(TIMER,BEGCLK)
        WRITE(2,30) REALRN
30      FORMAT(1X,F14.6)
C
C TURN OFF PRINTER CLOCK BY SENDING COMMAND 4 TO CLOCK I/O PORT
C
        CALL OUT(TIMER,STOPCK)
100     CONTINUE
        WRITE(5,40) MAX
40      FORMAT(1X,'NO. OF RANDOM NUMBER GENERATED =  ',I6)
        END

A.
```

Listing 3.

First prime number generator (Prime1) in BASIC

```
 10 REM PRIME PROGRAM 1 IN CDOS 16K BASIC
 20 REM MK/10-31-80
 30 PRINT"          BASIC PRIME NUMBER GENERATOR 1"
 40 INTEGER I,M,N,L
 50 PRINT
 60 INPUT"ENTER NUMBER OF PRIME NUMBERS DESIRED:  ",L
 70 PRINT
 80 PRINT"DIRECT PRIME NUMBER OUTPUT TO PRINTER"
 90 PRINT
100 OPEN\2,W\"$LP"
110 PRINT\2\"PRIME NUMBERS:"
120 PRINT\2\
130 PRINT\2\"               1"
140 PRINT\2\"               2"
150 REM L=NO. OF PRIME NUMBERS TO BE GENERATED
160 REM N=CURRENT NUMBER TO BE TESTED AS PRIME NUMBER
170 REM M=DIVISOR USED TO CHECK N TO BE PRIME NUMBER OR NOT
180 REM IF M IS NOT DIVISIBLE BY 2 THROUGH N-1, IT IS A PRIME NUMBER
190 REM I=NOL OF PRIME NUMBERS GENERATED SO FAR
200 I=2
210  FOR N=3 TO 32767.0
220   FOR M=2 TO(N-1)
230   IF N=(N/M)*M THEN GOTO 330
240   NEXT M
250  REM N IS NOT DIVISIBLE BY ANY NO. FROM 2 TO N-1, IT IS A PRIME NO.
260  REM TURN PRINTER CLOCK ON BY SENDING COMMAND 3 TO CLOCK PORT
270  OUT%00F1%,%0033%
280  PRINT\2\USING"################",N
290  REM TURN PRINTER OFF BY SENDING OUT COMMAND 4 TO CLOCK PORT
300  OUT%00F1%,%0034%
310  I=I+1
320  IF I=L THEN GOTO 340
330  NEXT N
```

(Continued on top of page 38)

```
C
C TURN ON PRINTER CLOCK
C
        CALL OUT(TIMER,BEGCLK)
C
        WRITE(2,140) N
C
C TURN OFF PRINTER CLOCK
C
        CALL OUT(TIMER,STOPCK)
C
C
C CHECK LIMIT OF PRIME NUMBER GENERATION
C
        NUMPRI=NUMPRI+1
        IF (NUMPRI.GE.MAX) GOTO 400
300     CONTINUE
400     WRITE(5,150) NUMPRI
150     FORMAT(1X,I6,'  PRIME NUMBERS GENERATED ON PRINTER')
        END
```

Listing 5.

Second prime number generator (Prime2) in BASIC

```
 10 REM PRIME PROGRAM 2 IN CDOS BASIC
 20 REM THIS PROGRAM IS FASTER THAN PRIME1 PROGRAM BY USING 3
 30 REM MODIFICATIONS SIMILAR TO PRIME2 FORTRAN PROGRAM
 40 REM MK/10-31-80
 50 PRINT"          BASIC PRIME NUMBER GENERATOR 2"
 60 INTEGER P(50),S
 65 REM FOR SPACE CONSIDERATION ONLY SAVE FIRST 50 PRIME NUMBERS
 80 INTEGER I,M,N,L
 90 PRINT
100 INPUT"ENTER NUMBER OF PRIME NUMBERS DESIRED:  ",L
110 PRINT
120 PRINT"DIRECT PRIME NUMBER OUTPUT TO PRINTER"
130 PRINT
140 OPEN\2,W\"$LP"
150 PRINT\2\"PRIME NUMBERS:"
160 PRINT\2\
170 PRINT\2\"               1"
180 PRINT\2\"               2"
190 REM L=NO. OF PRIME NUMBERS TO BE GENERATED
200 REM N=CURRENT NUMBER TO BE TESTED AS PRIME NUMBER
210 REM M=DIVISOR USED TO CHECK N TO BE PRIME NUMBER OR NOT
220 REM IF M IS NOT DIVISIBLE BY THE 2ND THROUGH THE KTH PRIME NOS.
230 REM OBTAINED EARLIER, IT IS A PRIME NO. K CAN BE SHOWN TO BE
240 REM THE SQUARE ROOT OF (N) - SEE DOCUMENTATION
250 REM I=NOL OF PRIME NUMBERS GENERATED SO FAR
260 I=2
270 REM INITIALIZE K (LIMIT FOR CHECKING DIVISIBILITY OF N)
280 K=1
285 P(1)=1
286 P(2)=2
290  FOR N=3 TO 32767.0 STEP 2
300  S=P(K)*P(K)
310  IF S<=N THEN K=K+1
320   FOR M=2 TO K
330   IF N=(N/P(M))*P(M)THEN GOTO 440
340   NEXT M
350  REM N IS NOT DIVISIBLE BY ANY NO. FROM 2 TO N-1, IT IS A PRIME NO.
360  REM TURN PRINTER CLOCK ON BY SENDING COMMAND 3 TO CLOCK PORT
370  OUT%00F1%,%0033%
380  PRINT\2\USING"################",N
390  REM TURN PRINTER OFF BY SENDING OUT COMMAND 4 TO CLOCK PORT
400  OUT%00F1%,%0034%
```

(Continued on next page)

Performance Evaluation

(Listing continued, text begins on page 34)

```
410   I=I+1
420   IF I<=50 THEN P(I)=N
430   IF I=L THEN GOTO 450
440   NEXT N
450 PRINT"NO. OF PRIME NUMBERS GENERATED  ",I
460 PRINT
470 PRINT
480 CLOSE\2\
490 BYE

>
```

Listing 6.

Second prime number generator (Prime2) in FORTRAN

```
C MK/11-2-80 PRIME2.FOR
C SECOND PRIME NUMBER GENERATOR WITH FASTER ALGORITHM COMPARED TO PRIME1.FOR
C THE MODIFICATIONS INCLUDE 3 MAJOR AREAS TO INCREASE THE SPEED;
C
C      1. IN THE DO 300 ..300 CONTINUE DO LOOP, DO 300 N=3,32767,2 IS USED
C         INSTEAD OF DO 300 N=3,32767 BECAUSE IT IS ONLY NECESSARY TO
C         CONSIDER ODD NUMBER AS PRIME NUMBER
C      2. THE CKLIMT IS USED SEE IF THE NUMBER BEING CONSIDERED IS DIVISIBLE
C         BY ANY NUMBER BETWEEN 2 TO CKLIMT.  IF THE NUMBER IS DIVISIBLE, IT IS
C         NOT A PRIME NUMBER.  CKLIMT WAS SET TO BE N-1 IN PRIME1 PROGRAM.
C         WIRTH (SYSTEM PROGRAMMING:AN INTRODUCTION,P137-141, PRENTICE-
C         HALL, 1973) HAS SHOWN THAT IT IS ONLY NECESSARY TO SET CKLIMT TO
C         THE SQUARE ROOT OF N.
C      3. IN THE DO 200..200 CONTINUE DO LOOP, THE DIVISOR ARE DERIVED
C         FROM THE PREVIOUS PRIME NUMBERS RATHER THAN FROM ALL NUMBERS
C         FROM 2 TO CKLIMT AS IN PRIME1 PROGRAM.  THE REASON IS IF N
C         WERE DIVISIBLE BY A NONPRIME K, THEN IT WOULD ALSO BE DIVISIBLE
C         BY THE PRIME FACTORS OF K.  IN PRIME2 PROGRAM, THE COMPUTED
C         PRIME NUMBERS ARE STORED IN AN ARRAY CALLED PRIM(1000).
C
C MK/10-28-80
C FORTRAN PRIME NUMBER GENERATOR
C
        PROGRAM PRIME
        INTEGER PRIM(1000),SQPRIM
        INTEGER TESTER
        INTEGER MAX, PRINUM, CKLIMT
        BYTE TIMER,BEGCLK,STOPCK
        DATA TIMER,BEGCLK,STOPCK/Z'F1',Z'33',Z'34'/
C
C TIMER = I/O PORT ADDRESS (HEX F1) FOR CLOCK SIMULATOR
C BEGCLK = COMMAND TO START PRINTER CLOCK (HEX 33 OR ASCII 3)
C STOPCK = COMMAND TO STOP  PRINTER CLOCK (HEX 34 OR ASCII 4)
C
C
C MAX = MAXIMUM NO. OF PRIME NUMBER TO BE CALCULATED
C PRINUM = PRIME NUMBER
C CKLIMIT= LIMIT FOR CHECKING PRIME NUMBER.
C THE CURRENT PRIME1.FOR PROGRAM THE CKLIMIT IS SET TO BE NUMBER-1.
C
C
        WRITE(5,100)
100     FORMAT (10X,'FORTRAN PRIME NUMBER GENERATOR',/)
        WRITE(5,110)
110     FORMAT(1X,'ENTER NUMBER OF PRIME NUMBERS DESIRED:  ')
        READ (5,120) MAX
120     FORMAT (I6)
        WRITE(5,125)
125     FORMAT(/,1X,'DIRECT PRIME NUMBER OUTPUT TO PRINTER',/)
        WRITE(2,130)
130     FORMAT(/,1X,'PRIME NUMBERS: ',/)
        PRINUM=1
        PRIM(1)=1
        WRITE(2,140) PRINUM
140     FORMAT (5I15)
        PRINUM=2
        PRIM(2)=2
        WRITE(2,140) PRINUM

C PRIME1.FOR USES A SLOW ALGORITHM TO GENERATE PRIME NUMBERS
C
C FOR GENERATION OF PRIME NUMBER FROM 3 TO MAXIMUM NO. (MAX),
C ALL ODD NUMBERS ARE TESTED TO SEE IF THEY ARE DIVISIBLE BY
C ANY NUMBER BETWEEN 2 AND CKLIMT, WHERE CKLIMT IS EQUAL TO
C THE MAGNITUDE OF THE NUMBER BEING CONSIDERED - 1.  THE NUMBER IS
C CONSIDERED AS PRIME NUMBER ONLY IF IT IS NOT DIVISIBLE BY ANY NUMBER
C BETWEEN 2 AND CKLIMT.
C
C IF THE NUMBER (N) IS EXACTLY DIVISIBLE BY ANY NUMBER BETWEEN 2 TO
C CKLIMT, IT IS DISCARDED AS BEING PRIME NUMBER BY EXITING THE LOOP
C BY THE GOTO 300 STATEMENT
C



C
C LET NUMPRI=NO. OF PRIME NUMBERS GENERATED SO FAR
C
        NUMPRI=2
        CKLIMT=1
        DO 300 N=3,32767,2
        SQPRIM=PRIM(CKLIMT)*PRIM(CKLIMT)
        IF (SQPRIM.LE.N) CKLIMT=CKLIMT+1
        DO 200 I=2,CKLIMT
        TESTER=(N/PRIM(I))*PRIM(I)
        IF (N.EQ.TESTER) GOTO 300
200     CONTINUE


C
C NUMBER NOT DIVISIBLE BY ANY NUMBER BETWEEN 2 TO CKLIMT, IT IS A PRIME
C NUMBER
C
C
C TURN ON PRINTER CLOCK
C
        CALL OUT(TIMER,BEGCLK)
C
        WRITE(2,140) N
C
C TURN OFF PRINTER CLOCK
C
        CALL OUT(TIMER,STOPCK)
C
C
C CHECK LIMIT OF PRIME NUMBER GENERATION
C
        NUMPRI=NUMPRI+1
C
C STORE PRIME NUMBER IN PRIM ARRAY
C
        PRIM(NUMPRI)=N
C
C CHECK IF ALL REQUIRED PRIME NUMBERS HAVE BEEN GENERATED
C
        IF (NUMPRI.GE.MAX) GOTO 400

300     CONTINUE
400     WRITE(5,150) NUMPRI
150     FORMAT(1X,I6,'  PRIME NUMBERS GENERATED ON PRINTER')
        END
```

End Listing

File Transfer Utilities

Send/Receive

by Lawrence Hughes

These routines are a follow-up to my article on MFT (*Dr. Dobb's Journal*, #49, October 1980, Vol. 5, #9), which was very well received. The original MFT utility was designed to solve the problem of transferring one or more files from one diskette to another on a single-drive system. These new utilities (SEND and RECV) are intended to solve a similar problem, namely transferring files from one CP/M system to another (with a different disk format), over a serial link. These are not intended to be suitable for file transfers over a phone line, for which I prefer my CLINK utility. These are intended to be used where the systems are physically adjacent (or at least within 50 feet of each other), and can be linked with an RS-232 cable. Baud rates of 9600 or above minimize the transfer time on large files or collections of files.

There are existing, commercially available utilities to perform this function but they are quite expensive and difficult to work with, since source code is not provided. As this is an essentially simple task, I have decided to release into the public domain a pair of programs (including source) that solve this problem in a general manner. Availability of source code makes it quite easy to bring them up on new systems. Note that the ability to program in assembly language, as well as information about port and status bit locations are required to move these routines to a new hardware system. Instead of a complex "installation" procedure, simple subroutines are provided that can be modified, and the entire programs assembled and loaded.

There are just three routines (INIT, initialize port; WACL, write ASCII character to link; and RACL, read ASCII character from link) that need be modified for any given system. Note that if the port in question can be initialized with a utility provided by the vendor (such as a CONFIG program), then the INIT routine can default to a simple RETURN. Note that both ends must be able to support 8-bit characters. Since the source code for the RECV program is just a few pages long, it can be typed in on the new system (minus comments), the I/O routines modified for that system, and used to transfer over the full source of both RECV and SEND. At that point, the I/O routines in the original source code can be modified for the new system, and you are up and running; files can be transferred in either direction.

Program Execution

To use these programs, execute RECV (no parameters) on the system which is to receive the new files, and then execute SEND on the system from which the files are to be sent. The argument of the SEND command may be a single file name, or an ambiguous file name (e.g., *.ASM). Files will be sent with checksums and automatic re-transmission, if needed, one sector at a time. Use the highest baud rate your hardware is capable of for minimum transfer time.

Note that these programs are the property of Mycroft Labs, and are hereby released into the public domain. They are not to be sold or included in a hardware or software package without the express permission of Mycroft Labs. Copies of the Mycroft Labs Public Domain disk (which includes MFT, SEND and RECV) are available for $15.00. Sorry, only 8" SS/SD "CP/M Interchange Format" is available. To order and/or receive permission to distribute any of these utilities, contact Lawrence E. Hughes at Mycroft Labs, Inc., P.O. Box 6045, Tallahassee, FL 32301, phone (904) 385-2708. In any case, the name "Mycroft Labs" must remain in the source code, and be displayed when the programs are executed.

(Listing begins on page 42)

SEND/RECEIVE

(Text begins on page 41)

Listing 1.

```
        title   'RECV - receive CP/M file from serial link'

;       author: Lawrence E. Hughes, Mycroft Labs
;       date:   80/06/07 - initial coding

;;;     port numbers and status bits for communications port
;
;       note: see routines INIT, RACL and WACL at end of program

data    equ     04H             ;serial port data
stat    equ     05H             ;serial port status
rda     equ     02H             ;read data avail. flag
tbe     equ     01H             ;trans. buffer empty

;;;     device independent equates
;

bdos    equ     0005H           ;BDOS entry point
wccfc   equ     02              ;write console character
wcbfc   equ     09              ;write console buffer
clsfc   equ     16              ;close disk file
delfc   equ     19              ;delete disk file
wsrfc   equ     21              ;write sequential record
makfc   equ     22              ;make disk file

sfcb    equ     005CH           ;system file control block
fn      equ     01              ;file name field
ft      equ     09              ;file type field
ex      equ     12              ;extent number field
nr      equ     32              ;next record field

dbuf    equ     0080H           ;system disk buffer

soh     equ     01H             ;start of header
stx     equ     02H             ;start of text
eot     equ     04H             ;end of transmission
enq     equ     05H             ;enquire
ack     equ     06H             ;acknowledge
bel     equ     07H             ;bell
lf      equ     0AH             ;line feed
cr      equ     0DH             ;carriage return
nak     equ     15H             ;negative acknowledge
etb     equ     17H             ;end of transmission block
can     equ     18H             ;cancel

eos     equ     '$'             ;end of string

        org     100H
recv:   lxi     h,0             ;save system SP
        dad     sp
        shld    oldsp
        lxi     sp,stack+64
        lxi     d,msg0          ;print heading
        call    wasc
        call    init            ;initialize link port

recv0:  call    racl            ;await ENQ from sender
        cpi     enq
        jnz     recv0
        mvi     a,ack           ;send ACK

        call    wacl
recv1:  call    racl            ;read char
        cpi     eot             ;exit if EOT
        jz      recvX

weolc:  mvi     a,cr
        call    wacc
        mvi     a,lf
        jmp     wacc

wacc:   push    h
        push    d
        push    b
        mov     e,a
        mvi     c,wccfc
        call    bdos
        pop     b
        pop     d
        pop     h
        ret

rmsg:   mvi     c,0             ;clear checksum
        push    h               ;save fwa of buffer
        push    b               ;save message length
rmsg1:  call    racl            ;read next char
        mov     m,a             ;store in buffer
        inx     h
        dcr     b               ;decrement count
        jnz     rmsg1           ;loop until zero
        call    racl            ;read checksum
        mov     a,c             ;jump if ok
        ora     a
        jz      rmsg2
        lxi     d,msg4          ;print 'Checksum error'
        call    wasc
        mvi     a,nak           ;request retransmission
        call    wacl
        pop     b               ;restore message length
        pop     h               ;restore fwa of buffer
        jmp     rmsg            ;go read retrans. message
rmsg2:  pop     b               ;exit
        pop     h
        ret


wasc:   push    h
        push    d
        push    b
        mvi     c,wcbfc
        call    bdos
        pop     b
        pop     d
        pop     h
        ret

close:  push    h
        push    d
        push    b

        mvi     c,clsfc
        call    bdos
        pop     b
        pop     d
        pop     h
        ret

delt:   push    h
        push    d
        push    b
        mvi     c,delfc
        call    bdos
        pop     b
        pop     d
        pop     h
        ret
```

```
        cpi     soh             ;jump if not SOH
        jnz     recv1
        lxi     h,sfcb+1        ;destination address
        mvi     b,11            ;message length
        call    rmsg            ;read filename
        xra     a               ;create file
        sta     sfcb+ex
        lxi     d,sfcb
        call    delt
        call    make
        cpi     255             ;jump if ok
        jnz     recv3
        mvi     a,can           ;abort sender
        call    wacl
        jmp     recvX           ;exit
recv3:  xra     a               ;rewind file
        sta     sfcb+ex
        sta     sfcb+nr
        lxi     d,msg6          ;print 'Now receiving
        call    wasc
        lxi     h,sfcb+1        ;print filename
        mvi     b,11
        call    pmsg
        call    weolc
        mvi     a,ack           ;send ACK
        call    wacl
recv4:  call    racl            ;read char
        cpi     etb             ;done if ETB
        jz      recvA
        cpi     can             ;jump if not CAN
        jnz     recv5
        lxi     d,msg2          ;print 'Transmission aborted'
        call    wasc
        jmp     recvA           ;close out file
recv5:  cpi     stx             ;jump if STX
        jz      recv6
        lxi     d,msg3          ;print 'Unknown message type'
        call    wasc
        mvi     a,can           ;abort sender
        call    wacl
        jmp     recvx           ;abort
recv6:  lxi     h,dbuf          ;capture sector
        mvi     b,128           ;record length
        call    rmsg            ;read sector
recv8:  lxi     d,sfcb          ;write buffer to disk
        call    write
        ora     a               ;jump if ok
        jz      recv9
        lxi     d,msg5          ;print 'Write error'
        call    wasc
        mvi     a,can           ;abort sender
        call    wacl
        jmp     recvx
recv9:  mvi     a,ack           ;accept message
        call    wacl
        jmp     recv4           ;await next block
recvA:  lxi     d,sfcb          ;close user's file
        call    close
        jmp     recv0           ;await next file
recvx:  lxi     d,msg7          ;print 'Done'
        call    wasc
        lhld    oldsp           ;exit to DOS
        sphl
        ret

pmsg:   mov     a,m
        inx     h
        call    wacc
        dcr     b
        jnz     pmsg
        ret
```

(Continued on top of page 42)

```
write:  push    h
        push    d
        push    b
        mvi     c,wsrfc
        call    bdos
        pop     b
        pop     d
        pop     h
        ret

make:   push    h
        push    d
        push    b
        mvi     c,makfc
        call    bdos
        pop     b
        pop     d
        pop     h
        ret



;;;     following device dependent code is for an IMSAI SIO2-2
;

;;;     init - initialize communications port
;

init:   lxi     h,inits
        mvi     b,6
init1:  mov     a,m
        inx     h
        out     stat
        dcr     b
        jnz     init1
        ret

inits:  db      00H,00H,00H,40H,6EH,27H

;;;     racl - read ascii character from link
;

racl:   in      stat
        ani     rda
        jz      racl
        in      data
        push    psw
        add     c
        mov     c,a
        pop     psw
        ret

;;;     wacl - write ascii character to link
;

wacl:   push    psw
wacl1:  in      stat
        ani     tbe
        jz      wacl1
        pop     psw
        out     data
        cma
        out     0FFH
        cma
        ret

msg0:   db      'RECV v2.0 - File Reception Utility - Feb 16, 1982',cr,lf
        db      'Copyright 1982, Mycroft Labs Inc.',cr,lf,cr,lf,eos
msg1:   db      'No disk space',cr,lf,eos
```

(Continued on next page)

SEND/RECEIVE

(Listing continued, text begins on page 41)

```
mss2:   db      'Transmission aborted',cr,lf,eos
mss3:   db      'Unknown message type',cr,lf,eos
mss4:   db      'Checksum error',cr,lf,eos
mss5:   db      'Write error',cr,lf,eos
mss6:   db      'Now receiving ',eos
mss7:   db      'Done',cr,lf,bel,eos

oldsp:  ds      2               ;system SP
stack:  ds      64              ;local stack

        end
```

LISTING 2.

```
        title   'SEND - Send CP/M file over serial link'

;       author: Lawrence E. Hughes, Mycroft Labs
;       date:   80/06/07

;;;     port locations and status bits for communications port
;
;       note: see routines INIT, RACL and WACL at end of program



data    equ     04H             ;serial port data
stat    equ     05H             ;serial port status
rda     equ     02H             ;read data avail. flag
tbe     equ     01H             ;trans. buffer empty flag

;;;     device independent equates
;


bdos    equ     0005H           ;bdos entry point
wccfc   equ     02              ;write console character
wcbfc   equ     09              ;write console buffer
opnfc   equ     15              ;open file
clsfc   equ     16              ;close file
srffc   equ     17              ;search first
srnfc   equ     18              ;search next
rsrfc   equ     20              ;read sequential record
sfcb    equ     005CH           ;system file control block
fn      equ     01              ;file name field
ft      equ     09              ;file type field
ex      equ     12              ;extent number field
nr      equ     32              ;next record field



dbuf    equ     0080H           ;system disk buffer

soh     equ     01H             ;start of header
stx     equ     02H             ;start of text
eot     equ     04H             ;end of transmission
```

```
        call    pmsg
        call    weolc



send6:  lxi     d,sfcb          ;read next record
        call    read
        ora     a               ;jump if ok
        jz      send3
        cpi     1               ;jump if eof
        jz      send7
        lxi     d,mss2          ;print 'File read error'
        call    wasc
        mvi     a,can           ;abort other end
        call    wacl
        jmp     sendx           ;exit
send3:  mvi     a,stx           ;send STX
        call    wacl
        lxi     h,dbuf          ;send record

        mvi     b,128           ;record length
        call    smsg            ;send message
        jc      send8           ;abort on error
        jmp     send6


send7:  mvi     a,etb           ;send ETB
        call    wacl
        lxi     d,sfcb          ;close file
        call    close
        lxi     h,flcnt         ;decrement filelist count
        dcr     m
        jmp     send0           ;loop
        jmp     sendx
send8:  lxi     d,mss7          ;print 'Aborted'
        call    wasc
sendx:  mvi     a,eot           ;send EOT
        call    wacl
        lxi     d,mss5          ;print 'All files sent'
        call    wasc
        lhld    oldsp           ;exit to DOS
        sphl
        ret



getfl:  lxi     h,flist         ;flptr = fwa of filelist
        shld    flptr
        xra     a               ;flcnt = 0
        sta     flcnt
        push    d
getfl1: call    srchf           ;search for first occurance
        cpi     255             ;jump if none found
        jz      getfl3
getfl2: ani     03H             ;de = dbuf + 32 * (a and 3)
        add     a
        add     a
        add     a
        add     a
        add     a
        mov     l,a
        mvi     h,0
        lxi     d,dbuf
        dad     d
        xchg
        lhld    flptr           ;hl = file list ptr
        xchg
        mvi     b,16            ;b = # of bytes to move
```

```
enq     equ     05H             ;enquire
ack     equ     06H             ;acknowledge
bel     equ     07H             ;bell
lf      equ     0AH             ;line feed
cr      equ     0DH             ;carriage return
nak     equ     15H             ;negative acknowledge
etb     equ     17H             ;end of transmission block
can     equ     18H             ;cancel


eos     equ     '$'             ;end of string



        org     100H
send:   lxi     h,0             ;save system SP
        dad     sp
        shld    oldsp
        lxi     sp,stack+64
        lxi     d,msg0          ;print heading
        call    wasc
        call    init            ;initialize link port
        lxi     d,sfcb          ;point to afn
        call    setfl           ;set file list
        lda     flcnt           ;jump if at least one found
        ora     a
        jnz     send0
        lxi     d,msg1          ;print 'File(s) not found'
        call    wasc



send0:  lhld    flptr           ;hl = ptr to next name
        lxi     d,sfcb          ;de = destination addr
        mvi     b,16            ;# of bytes to move
        call    move
        shld    flptr           ;update file list ptr
send2:  in      stat            ;await ACK from receiver
        ani     rda
        jnz     send2a
        mvi     a,enq           ;send another ENQ
        call    wacl
        jmp     send2



send2a: in      data            ;discard ACK
        lda     flcnt           ;exit if filelist count = 0
        ora     a
        jz      sendx
send1:  xra     a               ;open file
        sta     sfcb+ex
        lxi     d,sfcb
        call    open
        xra     a               ;rewind file
        sta     sfcb+nr
        mvi     a,soh           ;send SOH
        call    wacl
        lxi     h,sfcb+1        ;fwa of filename
        mvi     b,11            ;filename length
        call    smsg            ;send filename
        jc      send8           ;abort on error
        lxi     d,msg6          ;print 'Now sending'
        call    wasc
        lxi     h,sfcb+1        ;print filename
        mvi     b,11
```

(Continued on top of page 44)

```
        call    move
        xchg
        shld    flptr           ;update filelist pointer
        lxi     h,flcnt         ;increment filelist count
        inr     m
        pop     d
        push    d
        call    srchn           ;search for next occurance
        cpi     255             ;loop if another found
        jnz     setf12
setf13: pop     d
        lxi     h,flist         ;reset filelist pointer
        shld    flptr
        ret



pmsg:   mov     a,m
        inx     h
        call    wacc
        dcr     b
        jnz     pmsg
        ret



wcolc:  mvi     a,cr
        call    wacc
        mvi     a,lf
        jmp     wacc

wacc:   push    h
        push    d
        push    b
        mov     e,a
        mvi     c,wccfc
        call    bdos
        pop     b
        pop     d
        pop     h
        ret

move:   mov     a,m
        inx     h
        stax    d
        inx     d
        dcr     b
        jnz     move
        ret

srchf:  push    h
        push    d
        push    b
        mvi     c,srffc
        call    bdos
        pop     b
        pop     d
        pop     h
        ret



srchn:  push    h
        push    d
        push    b
        mvi     c,srnfc
        call    bdos
        pop     b
        pop     d
        pop     h
        ret
```

(Continued on next page)

SEND/RECEIVE

(Listing continued, text begins on page 41)

```
smsg:   mvi     c,0             ;clear checksum
        push    h               ;save fwa of message
        push    b               ;save message length
smsg1:  mov     a,m             ;fetch next char
        inx     h
        call    wacl            ;send it
        dcr     b               ;decrement count
        jnz     smsg1           ;loop until zero
        xra     a               ;send checksum
        sub     c
        call    wacl
        call    racl            ;await response
        cpi     ack             ;jump if ACK
        jz      smsg4
        cpi     nak             ;jump if not NAK
        jnz     smsg2
        pop     b               ;restore fwa and count
        pop     h
        jmp     smsg            ;send it again
smsg2:  cpi     can             ;jump if CAN
        jz      smsg3
        lxi     d,msg3          ;print 'Unknown code'
        call    wasc
smsg3:  pop     b               ;error exit
        pop     h
        stc
        ret
smsg4:  pop     b               ;normal exit
        pop     h
        stc
        cmc
        ret


wasc:   push    h
        push    d
        push    b
        mvi     c,wcbfc
        call    bdos
        pop     b
        pop     d
        pop     h
        ret


open:   push    h
        push    d
        push    b
        mvi     c,opnfc
        call    bdos
        pop     b
        pop     d
        pop     h
        ret


close:  push    h
        push    d
        push    b
        mvi     c,clsfc
        call    bdos
        pop     b
        pop     d
        pop     h
        ret
```

```
read:   push    h
        push    d
        push    b
        mvi     c,rsrfc
        call    bdos
        pop     b
        pop     d
        pop     h
        ret


;;;     following device dependent code is for IMSAI SIO2-2
;
;;;     init - routine to initialize communications port
;
init:   mvi     b,6
        lxi     h,inits
init1:  mov     a,m
        inx     h
        out     stat
        dcr     b
        jnz     init1
        ret

inits:  db      00H,00H,00H,40H,6EH,27H

;;;     racl - read ascii char from link (do not strip bit 7)
;
racl:   in      stat
        ani     rda
        jz      racl
        in      data
        ret

;;;     wacl - write ascii char to link
;
wacl:   push    psw
wacl1:  in      stat
        ani     tbe
        jz      wacl1
        pop     psw
        out     data
        cma
        out     0FFH
        cma
        add     c
        mov     c,a
        ret

msg0:   db      'SEND v2.0 - File Transmission Utility - Feb 16, 1982',cr
        db      'Copyright 1982, Mycroft Labs Inc.',cr,lf,cr,lf,eos
msg1:   db      'File(s) not found',cr,lf,eos
msg2:   db      'File read error',cr,lf,eos
msg3:   db      'Unknown response',cr,lf,eos
msg5:   db      'Done',cr,lf,bel,eos
msg6:   db      'Now sending ',eos
msg7:   db      'Aborted',cr,lf,eos

oldsp:  ds      2               ;system SP
stack:  ds      64              ;local stack
flcnt:  ds      1               ;file list count
flptr:  ds      2               ;file list pointer
flist:  ds      16*64           ;file list

        end
```

End Listing

Numeric Data Processor

The Intel 8087

The Intel 8087 is an astonishingly powerful numeric co-processor for 8086/8088 microcomputers. It implements the proposed IEEE floating-point standard on a single integrated circuit. The 8087 is an order of magnitude faster than its forerunners (see Table 1) and incorporates a degree of precision and sophisticated exception handling that surpasses many mainframe computers.

The 8086 was announced over three years ago, but due to production difficulties the chip has only become available off-the-shelf in the last few months. Delivery delays aside, the high price (upwards of $450) frightened off most of the experimenters who form the cutting edge for the interface and software exploitation of new IC's. Then a funny thing happened . . . IBM started selling a personal computer, built around the Intel 8088 microprocessor, that was found to have an undocumented, empty 40-pin socket on the motherboard. Within a matter of hours, it became an open secret in microcomputer circles that the mystery socket was properly wired to accept an 8087 numeric processor. There was an immediate upsurge of interest in the 8087 among scientists, engineers, speed freaks, and of course hungry software developers.

We will try to provide here a brief survey of the architecture and capabilities of the 8087 co-processor, and some warnings as well. But first, let us observe a moment of silent admiration for the pioneer IBM PC owner who, on the basis of faith and hearsay, plugged an 8087 (literaly worth more than its weight in gold) into that famous empty socket and turned on the juice.

Overview

The 8087 is constructed with HMOS technology, and contains the equivalent of over 65,000 devices on a single substrate. It is packaged in a 40-pin DIP and operates from a single +5 volt power supply. The numeric algorithms are embodied in some 30,000 bits of microcode.

Previous numeric processors for microcomputers, such as the AMD 9511 and AMD 9512, could be interfaced to practically any microprocessor with a minimum of fuss, and were handled by the software simply as other peripheral devices. In contrast, the 8087 operates in an extremely tightly coupled configuration with the host 8086 or 8088, sharing the same instruction stream and many control signals, and can not conceivably be made to work with any other microprocessor.

When a program is created that will utilize the 8087 numeric data processor (NDP), the commands are assembled in line with the instructions executed by the 8086/8088 central processing unit (CPU). All control codes for the NDP begin with a characteristic "Escape" byte, followed by one or more bytes which describe the operation to be performed and the location of the operands. As the CPU fetches and decodes instructions, the NDP monitors the address and data lines, ignoring any instructions that do not apply to it. When an Escape byte is encountered, the CPU proceeds to fetch the following bytes in the normal way, which gives the NDP the opportunity to grab and examine the entire instruction sequence for its internal use. If the instruction contains an embedded reference to a memory operand, the CPU will also calculate the effective address and place it on the system bus; the NDP can capture the address, take control of the bus, and perform the necessary data transfer.

This scheme, where only one processor is responsible for fetching the instruction stream while each co-processor picks out the commands that it can execute, is both elegant and efficient. The need for transfer of control information between processors is minimized, since all have access to the same information. Memory address calculation logic can be localized in one processor. Lastly, there is little contention over the bus, since each processor is not trying to fetch its own instruction stream.

The pooled resources of the CPU and NDP are formidable (see Figure 1). The 8086/88 has eight semi-general-purpose registers, four memory segment registers, a status word, and an instruction pointer. The 8087 adds two control and status words and eight 80-bit, floating-point (FP) registers. The FP registers may be addressed explicitly or the entire set used as a pushdown stack. The latter mode simplifies subroutine programming, since data may be referenced relative to the current top of stack without concern for its actual physical location.

In general, the CPU/NDP combination may be viewed and programmed as a single unit, with the advantage that operations on each processor may proceed concurrently. The major restriction is that communication between the two processors must take place via shared memory; there can be no direct transfer of data from the registers of one processor to the other. To forestall the possibility of one processor trying to fetch a data value from memory before the other had finished writing it, a special "Wait" instruction is provided that can be used to force synchronicity.

Data Formats

The 8087 recognizes three classes of data types: packed decimal integers, binary integers, and binary real (floating point) numbers. The two binary classes are further subdivided into long and short variations (see Figure 2). All of these data formats exist in memory only. When data is loaded into an 8087 register, regardless of its original type, it is converted into a format called "temporary real" which is used for all internal operations. A temporary real number is 80 (!) bits in length and consists of a sign bit, a 15-bit biased exponent, and a 64-bit significand (sometimes called the mantissa or fraction). Its extra length shields the user from cumulative rounding errors or overflow/underflow in intermediate calculations.

An interesting byproduct of this approach is that the FIX and FLOAT operations of other numeric processors are no longer necessary. To convert a number from one data type into another, you merely load the number into a NDP register, then issue a command to write it back to memory in the desired format.

8087 Instruction Set

The 8087 has 68 instructions which may be broadly divided into six groups: data transfer, arithmetic, comparison, transcendental, constants and control.

Each instruction is composed of two to four bytes, depending on the location and addressing modes of the operands. Each byte is in turn broken up into bit fields that specify the operation, index registers, displacement, floating-point source or destination registers, and in some cases the format of the number as it is stored in memory. See Table 2, which gives a summary of the 8087 commands.

by Ray Duncan

Ray Duncan, Laboratory Microsystems, 4147 Beethoven Street, Los Angeles, CA 90066.

Control and Error Handling

The 8087 provides the programmer with unprecedented control over the way numeric operations are performed. Four modes of rounding are available: round to nearest, round up, round down, and "chop." Results may be calculated with 64, 53, or 24 bits of precision (64 is the default, the others are for result compatibility with previous floating-point packages and do not offer any speed advantage). The programmer may select between projective closure (all infinite results are treated identically) and affine closure (negative infinity is differentiated from positive infinity). All of these options may be selected or changed at execution time by modifying the NDP control word.

During numeric operations, six classes of exception or error conditions are reported back to the host CPU: invalid operation, overflow, underflow, zero-divide, denormalized number, and precision (advising that the result could not be represented exactly and rounding was required). Each type of exception sets a bit in the NDP status word, and may also cause an interrupt to the CPU depending on the mask bits in the NDP control word. By modifying the control word the programmer may accept responsibility for handling certain exceptions and delegate the others to the NDP. For any masked exception, the NDP's default fixup responses have been tailored to deliver the most "reasonable" result.

Programming the 8087

Having caught your eye with that heading, I must now admit that programming the 8087 numeric processor is not a subject that can possibly be covered in the space allotted for this column. This is not a situation where you can plug in a chip, whip a few patches into your BASIC interpreter, and run rings around the competition. The many powerful operations of the 8087, when combined with the various addressing modes of the 8086/88, result in an instruction set complex enough to give pause to even the most dedicated hacker. The intricacies of assembling those crazy 8086 address fields, using the NDP efficiently, avoiding deadlocks and race conditions with the host CPU, and handling asynchronous exception traps are simply going to put machine-level programming of the 8087 outside the limits of time and patience available to the average personal computer user. I strongly advise that you sit back and let the high-level language implementors get the headache instead.

If you are really interested, the application notes and manual cited at the end of this article provide some excellent tutorial material and detailed programming examples; however, the latter won't do you much good without the $1250 Intel 8087 software support package and an Intel development system to run it on.

Problems and Future Directions

Over the past year, many rumors have been circulating concerning the 8087's propensity to overheat and its finicky timing characteristics. Until recently, Intel was only selling the 8087 in matched and tested pairs with an 8088 and there was an eight-week lead time on delivery. These problems appear to have been overcome, and you can now pick up an 8087 at will from Hamilton-Avnet and other distributors. Only a 5 mHz version is available at present, so the 8 mHz 8086 and 8088 owners are still out of luck.

The cost of an 8087 remains as a major stumbling block. The current sticker price of $450+ is a powerful deterrent to experimenters and software developers. On the positive side, a combination of progress on the production learning curve and second-source manufacturers is expected to bring the cost per chip down to

	Intel 8087 5 mHz	8086 machine code emulation	AMD 9512 3 mHz
64-bit multiply	27	2100	590
64-bit divide	39	3200	1640

Table 1.
Comparative execution times (in microseconds) for real number operations.

Data Transfer
Fetch number from memory in any format, convert to temporary real, and load into floating-point register. Exchange any two floating-point registers. Convert contents of floating-point register into desired data type and store into memory. Optionally push the stack on a load from memory or pop the stack on a store.

Arithmetic
Add, subtract, subtract reversed, multiply, divide, divide reversed, square root, scale, modulo division, round to integer, absolute value, change sign. Optionally pop the floating-point register stack after operation complete.

Comparison
Compare two floating-point registers, or a floating-point register and a memory operand. Alternatively, examine a single register or memory operand and set the status flags accordingly. Optionally pop the floating-point register stack to discard one or two operands.

Transcendental
Tangent, arctangent, powers, and logs to the base 2.

Contants
Load floating-point register with zero, one, pi, common or natural log of two, or base two log of the constant e.

Processor control
Load or store control word, store status word, load or store 8087 environment and/or register set, enable or disable interrupts, clear exceptions, initialize processor.

Table 2.
Intel 8087 Numeric Data Processor instruction repertoire.

the neighborhood of $150 by the end of 1983. Whether even this price will be low enough for the 8087 to win widespread use is uncertain. The AMD 9511 and 9512 have been selling in this range for several years, but their use in microcomputer applications remains uncommon.

Another barrier to acceptance of the 8087 is certainly going to be lack of software development tools. Past history is again discouraging: the AMD 9511 and 9512 have not received any significant attention from high-level language vendors (with the exception of one Pascal compiler, one Fortran subroutine library, and one Forth system that I know of). However, the predictable machine environment of the IBM PC, plus the incentive of the estimated one million machines that will be in the field within five years, may result in the appearance of many language packages that would have never been written otherwise.

Further Reading

Intel iAPX 86,88 User's Manual and Numeric Supplement, August 1981. $10.00 from Intel Literature Department, 3065 Bowers Avenue, Santa Clara, California 95051.

Getting Started with the Numeric Data Processor, Intel Application Note AP-113. Free when you order the User's Manual.

Making Mainframe Mathematics Accessible to Microcomputers. Intel Article Reprint AR-135. Free when you order the User's Manual. ▶▶J

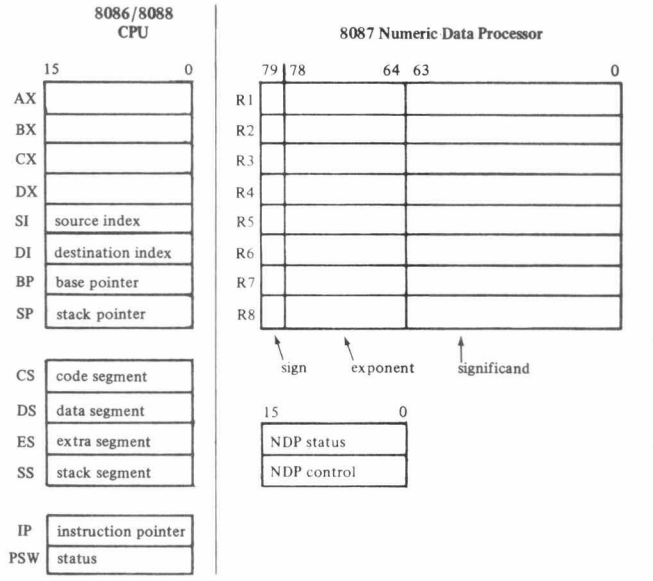


Figure 1.
The 8086/8088 CPU and 8087 numeric co-processor register sets.

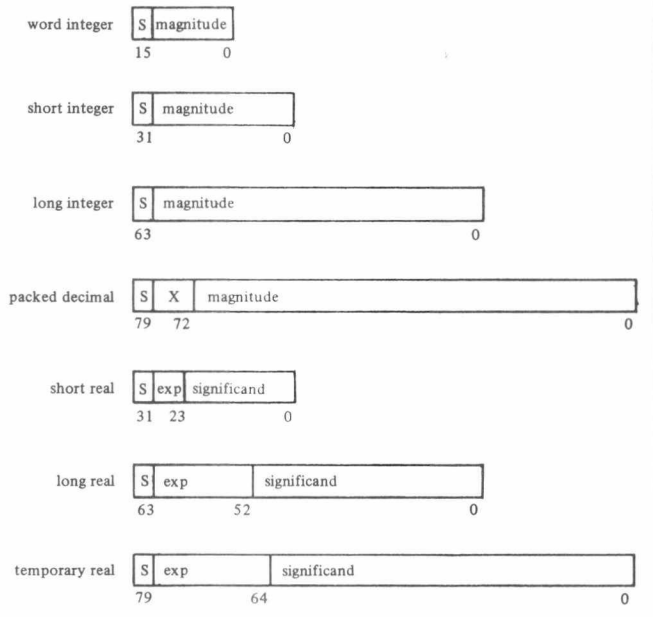


Figure 2.
The 8087 numeric data formats. S is sign bit. Exponent bias: 127 for short real, 1023 for long real, 16383 for temporary real. Short and long real significands are left normalized with an implicit integer bit.

Apple-Verter, Model APX-800, is a high-fidelity color video modulator that connects Apple II and other computer systems to a home TV rather than a costly color video monitor. Made by ATV research, and selling for $29.75, the Apple-Verter operates above normal computer harmonics, in the high VHF band (tunable channels 7-10), giving it, they claim, "unusually high stability as compared to some UHF units." The unit mounts inside the computer and plugs into the existing power/video plug, and attaches to the TV with a ten-foot direct-connect antenna cable **(Reader Service No. 335).**

Microconnection for the Osborne 1 is a Bell 103-compatible modem that operates in both originate and answer mode at 300 baud. A cassette recorder can be plugged in to transcribe on-line communications for later playback. Microconnection weighs less than a pound, and measures 7.7 by 5.5 by 1.7 inches. Made by Microperipheral Coporation, it costs $159 **(Reader Service No. 345).**

Inmac's **direct-connect modem** connects any phone line by an RS-232 interface to a mini- or microcomputer or peripheral device. It is Bell 103-compatible, operates at 300 baud, and automatically switches between answer and originate modes. The modem uses full duplex, asynchronous data communication modes, and features self-diagnostics, a carrier-detect LED, and push-button switching between voice and data communications. It weighs less than two pounds, fits under the telephone, and costs "under $200" **(Reader Service No. 355).**

The **Mimic** speech processor is a low-cost audio signal processing unit, offering speech processing for several popular microcomputers. Packages range from bare boards for user assembly to complete systems. **Mimtalk**, a BASIC program, outputs speech from prepared vocabularies of phonemes, words, or phrases, and **Mimhear** is an automatic speech recognition program. Prices range from under $20 for the board to $199 for a complete system. Mimtalk costs $34.95 **(Reader Service No. 365).**

The **Apple Speech Computer Board (SCB-APPLE)**, from Multitech Electronics, is based on Texas Instruments' TMS 5200 speech-synthesis device. It plugs directly into any spare slot of an Apple II, and has an on-board audio amplifier providing a driver for the 8-ohm Apple II speaker. A socket is provided for TI's TMS 6100 voice synthesis memory for expansion. The system includes a mini-floppy with speech data and a 400-word English vocabulary, and three Chinese lessons. Other languages can be developed, including male and female voice inflections. ($129. **Reader Service No. 375).**

Microsoft's **RAMCard** for the IBM Personal Computer offers up to 256K additional memory, and disk emulation capability, using some or all of the memory as RAMDrive for disk I/O. RAMCard can be used in any slot of the PC, and up to three can be installed simultaneously, for up to 480K additional memory. Configurations are 64, 128, 192, and 256K, ranging in price from $495 to $1095, and a 64K upgrade kit for the lower configurations, $200 **(Reader Service No. 385).**

Do you want a good-looking printer? How about inexpensive? The **Alphacom 40** won the "Excellence of Design" award from *Industrial Design* magazine. This 40-column, high-speed (four lines per second), thermal printer with graphics capability costs $295. The **Alphacom 42**, a two-lines-per-second version, costs $195 **(Reader Service No. 395).**

Sixteen-Bit Systems

Sixteen-bit systems are coming down in price. Computex's Z8001-based **X-8000**, in a typical configuration (each system can be tailored to the user's requirements), has 256K dynamic RAM, expandable to 16 megabytes, dual eight-inch floppy drives, a 15-slot motherboard, eight serial I/O ports, the ZMOS operating system, supporting up to 32 users simultaneously (each running a different program), paged-segmented memory circuitry, sockets for two 2716 EPROMs, eight vectored interrupt levels plus a non-maskable interrupt input, two programmable timers, a socket for optional 9511 arithmetic processor, a cast aluminum case and power supply and costs $7053. The 68000-based **X-6000**, is fully compatible with Intel's Multibus, handles up to 64K of onboard PROM and 128K or onboard RAM, and has seven prioritized vectored interrupts and one non-maskable interrupt, two RS-232 serial ports with 40 bits of parallel I/O, and a triple 16-bit timer/counter for generating interrupts or monitoring external events. In a similar configuration to the X-8000 just described, and with an RMX-86-like operating system, an X-6000 system costs $7099. Languages include BASIC, SuperSoft C, COBOL, FORTRAN, Pascal, and Ada **(Reader Service No. 415).**

Digital Microsystems offers its **System 5000** in either eight- (Z80-A CPU) or 16-bit (8086 CPU) configuration. Both have an 8086 video controller. The high-resolution, 15-inch, nonflicker (P-39 phosphor) display can be mounted either vertically, to function as a full-page word processor displaying 80 characters by 66 lines, or horizontally, with 80 columns by 26 lines of high-resolution characters, or as a spread-sheet-type screen of 132 columns by 50 lines. The System 5000 has a detachable keyboard and 30 programmable function keys. Screen intensity can be changed and inverse video invoked under program control or by control key sequences. The stand-alone workstation can be incorporated into the HiNet local area network, a CP/M-based packet-switched network, with availability of all CP/M software, and supporting up to 32 users. The system is fully bit-mapped, with a horizontal scan rate of 19,200 hertz, an interlaced display of 64 hertz, and a display format of 800 by 560 pixels. High-resolution characters are 8 by 17 dots, in a cell size of 10 by 21. The eight-bit System 5000 starts at $3295, and the 16-bit System 5000 at $4195 **(Reader Service No. 425).**

Operator provides automatic dialing and answering for computer owners who now manually dial and answer phone calls using acoustic couplers or manual, direct-connect modems. Features include baud rates from 110 to 300, half- and full-duplex modes, fast dial (20 pulses per second), carrier monitor, command interpreter, and command-prompting output. Operator can be in one of two modes, terminal (passing characters directly from computer to modem unchanged) and command (processing input). The unit comes with complete documentation, and costs $129 **(Reader Service No. 445).**

Conect software from Vector Graphic provides asynchronous data communications in any of four basic operating modes for all Vector single- and multi-user microcomputer systems. In send/receive mode, data files can be transmitted between any two Vector systems, with automatic error detection, and retransmission if necessary. The link mode transfers files or programs between computers, permitting remote diagnostics and implementation of bulletin boards on

Vector systems. Handshaking protocols permit communication with minis and mainframes. The terminal mode emulates the LSI ADM3A or Hazeltine 1500 dumb terminal, turning the microcomputer into an interactive communications terminal that can send or receive data at up to 1200 baud. In the interactive mode, a Vector microcomputer can communicate, at 300 or 1200 baud, with minicomputers or mainframes without special software at the host end. In this mode, a Vector microcomputer can also communicate with other Vector systems or the LSI ADM3A or Hazeltine 1500. In the interactive mode, the Vector system can use a modem to access public data bases, with automatic dialing and answering for some modems and a directory of telephone numbers that can be automatically dialed. Other options include selectable originate/answer mode, full- or half-duplex operation, auto-repeat dialing and answering, and voice communications interrupt. Conect software costs $150 **(Reader Service No. 455)**.

One of the services you can access with some of this wonderful equipment is **NewsNet**, which electronically transmits over 100 newsletters from 30 publishing groups, including *Communications Daily*, *Satellite Week*, and *Telephone News*, and publications in the energy, government, environment, educational, and business fields. There is no initiation fee. Cost is $24 per hour, with a $15 per month minimum **(Reader Service No. 465)**.

A software vendor directory lists software compatible with Hayes products, including the Micromodem II, Micromodem 100, Hayes Stack Smartmodem, and Hayes Stack Chronograph. Software for most popular computers includes programs that automatically answer phone calls and determine transmission speed, automatically switch between host and terminal modes, redial if necessary, make outgoing calls at preset times, automatically log on, send and receive ASCII files, terminate the call, request passwords, communicate between any two systems, access PRESTEL, turn a microcomputer into an asynchronous terminal emulator, copy incoming data into a 30K buffer for later transfer to disk, play chess, Go-Moku, or backgammon between remote partners, emulate smart terminals, do word processing, access Western Union's services, permit users to access their own computers by phone, time a call and maintain a log for comparison with phone bills, transfer high-resolution pictures between computers, and much more **(Reader Service No. 475)**.

The **JobStream CP/M 2.2** Operating System, from Aton International, for TRS-80 Model II and Model 16 increases the average speed of diskette operations up to five times, by buffering several disk tracks in RAM, so that "floppies are no longer the major limiting factor in system efficiency." User programs can be as large as 62K, and, if the program accesses other 32K memory boards for data and program code storage, over 200K. JobStream permits Model II users to upgrade to CP/M. Level I operates in the standard 64K memory and costs $179. Level II requires 96K or more and costs $235. An upgrade kit, from Level I to II, including diskette and documentation, is $81. They also offer BASIC Surrogate, and VisiCalc Surrogate, for $99 apiece, or $75 each if ordered with JobStream, to allow Radio Shack BASIC and VisiCalc to operate in the JobStream CP/M environment. OmniWriter, $150 (or $125 with JobStream) is a video text editor that can edit two files simultaneously, scroll forward or backward by line or page, and perform other word processing functions. Safety features include interrupting any automatic process with one keystroke, and holding deleted copy in a temporary buffer for recall of inadvertently deleted text **(Reader Service No. 495)**.

The **Floppy Disk Librarian** from Little Bit helps the IBM Personal Computer keep track of floppy disks and files. A consolidated catalog displays name, date, size, and comments for all disks and all files, and what is where. Reading the directory tracks of a disk automatically updates the Librarian. Information can be retrieved by disk or file name, and wildcarding can be used. Output can be sent to display, printer, or a disk file. $39.95 includes a manual and on-line help files **(Reader Service No. 505)**.

Data Perfect is a spreadsheet program for Atari or Apple that allows users to design their own screen masks. It has utilities, a report generator, and a mailing label generator. It supports multiple searches and sorts. Complete formula and mathematical operations can be performed on and between fields. The program supports one or two disk drives, can be used with any parallel printer, and requires 32K. LJK Enterprises offers the program for $99.95 **(Reader Service No. 308)**.

ZenCalc is a spreadsheet caclulator for Heath/Zenith computers. It does financial and numerical model building, forecasting, and problem solving, has a two-dimensional "What if?" calculator program, includes formulas, labels, and variable column width, and outputs to printer or disk file. Tailored to the H89/Z89 computer or the H19/Z19 terminal, cursor control and function keys "provide a true full-screen editor 'feel' in worksheet manipulations." The manual includes a step-by-step tutorial. ZenCalc comes with examples, including templates that automatically compute the full IRS 1040 form and itemized deduction Schedule A. The Software Toolworks offers the program for $99.95 – plus $2 p and h **(Reader Service No. 318)**.

The Software Toolworks also offers several programs for the Osborne 1, which are priced from $19.95 to $49.95. **Spell** is a 50,000-word proofreader for WordStar files. **MyChess** is a championship level chess program with a full graphics board display. "Original" **Adventure** is available. They have the **Eliza** "therapist," and arcade-style video games. Programming languages include C, RATFOR, LISP, and macroassemblers for Z80 and 8080. They offer a free software catalog **(Reader Service No. 328)**.

In a description of her **Handy Home Hints** program, Nancy Modney says, "Get your wife involved with your microcomputer." That sounds to me like reverse sexism: what if the lady of the house is also the programmer? Anyway, the cassette tape "teaches you 50 unique ways to save time and money in areas such as energy costs, cooking, cleaning, child care, and home maintenance." System requirements are TRS-80 Model I or II level II with at least 16K. Nancy offers free details for an SASE. Nancy Modney, 4144 N. Via Villas, Tucson, AZ 85719.

macroP adds macro commands to a document formatter or language processor. For assembly language, it converts simple assemblers into macro assemblers. For high-level languages, it permits indefinitely long variable names, compile-time expression evaluation, conditional compilation, and time and data stamping of source programs. For text formatting, macroP adds conditional nested inclusion of text files and automatic numbering of section headings. The package includes a manual and files of macros tailored to

BASIC, PL/I, Pascal, and to Textwriter. Available for IBM PC-DOS, CP/M, and CP/M-86 from Pluto Research, macroP costs $135 **(Reader Service No. 338).**

Languages

basic′ (basic prime) is a "supremely logical programming environment" from Delta Micro Systems. It is a structured extension of BASIC. Blocks of code are indicated by a unique indentation convention, eliminating the need for extraneous statements (such as BEGIN and END) and statement numbers. Comments are right justified, block indentation reveals the flow of control, procedures are separated by white space, and an extensive cross reference is provided. A basic′ program is written with editor′ (supplied), a full-function text file editor optimized for programming, and is then translated into a normal BASIC program, producing a listing that cross-references the basic′ program to its BASIC translation. basic′ comes with documentation and program diskette in a binder, is available for Apple, IBM PC, and TRS-80, with other systems under development, and costs $129 **(Reader Service No. 348).**

I've saved the best bargain for last. You can get **JRT Pascal**, a Pascal compiler for CP/M, from JRT Systems for $29.95. They'll let you copy the system and the manual and give them to friends, as long as you don't resell them. The package includes a one-pass compiler that does not need linking or macroassembly, Exec, the run-time environment, with dynamic storage, floating-point arithmetic, and random disk support, an assembler, an optional linker that merges Pascal or assembly external procedures and functions into one module, Customiz, to customize the Pascal system to individual hardware and software environments, and Activan, an activity analyzer that interacts with a program to monitor execution frequency by line number. printing a histogram to shown activity in each program area. External procedures are also included that compute statistical function on an input array, format and print X-Y graphs, scatter graphs, and histograms, and print large block letters for attention-getting report headers. Fourteen digits of precision binary-coded-decimal (BCD) format eliminates conversion errors, with a floating-point exponent range from -64 to +63. Strings can have up to 64K characters, not just the 255 limit of other versions of Pascal. Records contain logically related data items, with key indexing. Random access to disk files is by both relative record number and relative byte address. Arrays can be formed from basic data types and from user-defined types. Indexes are user-defined and can be integers or characters. Arrays can be up to eight dimensions. You need an eight-inch, single-density disk drive and a 52K CP/M system **(Reader Service No. 358).** DDJ

Clinic *(Continued from page 9)*

V. Pitteway, "Algorithm for Drawing Ellipses or Hyperbolae With a Digital Plotter," *Computer Journal*, 10(3):282-289, November 1967). Pitteway's method will draw any ellipse using only addition and subtraction.

The problem of finding the ellipse was solved by a number of *DDJ* readers. Almost all the solutions were equivalent. Most readers used one of two methods to find the solution. Either they treated the ellipsoid as a deformed unit sphere, or they took the formula for the rotated ellipsoid directly and used partial derivatives to find lines tangent to its surface. One of the first solutions to arrive was provided by Bruce Kendall. He showed that any ellipsoid could be represented by a product of matrices:

$$\mathbf{X} = \begin{bmatrix} r\cos(\phi) \\ r\sin(\phi) \\ z \end{bmatrix}$$

$$\mathbf{P} = \begin{bmatrix} \frac{1}{a^2} & 0 & 0 \\ 0 & \frac{1}{b^2} & 0 \\ 0 & 0 & \frac{1}{c^2} \end{bmatrix}$$

$$\mathbf{X}^t \mathbf{R}_{yz}^t \mathbf{R}_{zx}^t \mathbf{R}_{xy}^t \mathbf{P} \mathbf{R}_{xy} \mathbf{R}_{zx} \mathbf{R}_{yz} \mathbf{X} = 1$$

where R is the rotation of the axis matrix for each axis, and a superscript t denotes a matrix transpose.

Kendall showed that a solution resulted from taking the partial derivative with respect to z and setting the result equal to zero. If the ellipsoid resulting from the rotation of axes is expressed as

$$\mathbf{Q} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

such that

$$\begin{bmatrix} x & y & z \end{bmatrix} [\mathbf{Q}] \begin{bmatrix} x \\ y \\ z \end{bmatrix} = 1$$

the projection of the ellipsoid (its shadow as cast by an infinitely distant light source) is

$$\begin{bmatrix} x & y \end{bmatrix} \left(\begin{bmatrix} a & b \\ c & d \end{bmatrix} - \frac{\begin{bmatrix} c \\ f \end{bmatrix} \begin{bmatrix} g & h \end{bmatrix}}{} \right) \begin{bmatrix} x \\ y \end{bmatrix} = 1$$

While this is a very compact way of expressing the solution, it might not be obvious what it means. If your ellipsoid is centered at the origin and, prior to rotation, extends from $-a$ to a along the x-axis, from $-b$ to b along the y-axis, and from $-c$ to c along the z-axis, it is said to be in "standard form." If you are on the z-axis, far from the origin, it will appear as an ellipse in standard form from $-a$ to a, $-b$ to b. If you now rotate the ellipsoid about the x-, y-, and z-axes by r, s, and t degrees respectively, the ellipsoid will still appear as an ellipse, though possibly not in standard form, at the origin. This solution gives the formula describing that ellipse. The solution, completely expanded and ready to use, is shown in the box below.

To use the formula, substitute the known values for a, b, and c (the size of your ellipsoid), and plug in the sin and cosin of the angles by which it is rotated around the three axes. The result will be a constant times X^2 plus a constant times Y^2 plus a constant times XY, equated to 1. Drawing this formula on the screen will show how the ellipsoid would appear to an observer who is far away on the z-axis.

By this point you may have realized that this method is not going to give us the order of magnitude increase in speed that was the original goal. To find the projection of the ellipsoid will require roughly fifty multiplications, three times the number needed to find the projection of a point. In addition, clipping along the edge of the screen is no longer a trivial task; it now requires solving quadratic equations. The hidden line problem is made both easier and harder, in that there are about a third as many lines cluttering up the screen but finding out what is hidden looks much more difficult.

At this point it looks as if this method, with a lot of work, could be as fast as the point-transformation method, but with a more complicated program. Given the number of points that a single ellipsoid might represent, the computation time required would be roughly equal.

I would like to thank every one who took the time to offer their assistance in solving the problem. I have not given up my search, and am now looking for another way that might be used to gain the order of magnitude increase in speed that I need.

– Don Taylor

Don also sent a program in BASIC that used the formula to draw outlines of

Expanded Ellipsoid Solution

$$K=-\sin(t)\cos(r)+\cos(t)\sin(s)\sin(r)$$

$$L=\sin(t)\sin(r)+\cos(t)\sin(s)\cos(r)$$

$$M=\cos(t)\cos(r)+\sin(t)\sin(s)\sin(r)$$

$$N=-\cos(t)\sin(r)+\sin(t)\sin(s)\cos(r)$$

$$P=\frac{\sin^2(s)}{a^2}+\frac{\cos^2(s)\sin^2(r)}{b^2}+\frac{\cos^2(s)\cos^2(r)}{c^2}$$

$$Q=\frac{-\cos(t)\cos(s)\sin(s)}{a^2}+\frac{\cos(s)\sin(r)K}{b^2}+\frac{\cos(s)\cos(r)L}{c^2}$$

$$R=\frac{-\sin(t)\cos(s)\sin(s)}{a^2}+\frac{\cos(s)\sin(r)M}{b^2}+\frac{\cos(s)\cos(r)N}{c^2}$$

$$X^2\left(\frac{\cos^2(t)\cos^2(s)}{a^2}+\frac{K^2}{b^2}+\frac{L^2}{c^2}-\frac{Q^2}{P}\right)+Y^2\left(\frac{\sin^2(t)\cos^2(s)}{a^2}+\frac{M^2}{b^2}+\frac{N^2}{c^2}-\frac{R^2}{P}\right)+2XY\left(\frac{\cos(t)\sin(t)\cos^2(s)}{a^2}+\frac{MK}{b^2}+\frac{LN}{c^2}-\frac{QR}{P}\right)=1$$

a rotated ellipsoid. We converted his code to run on our Atari. The program appears in Listing 3, below. When the program asks for the X, Y, and Z coefficients, give it input of 20, 40, 60. That defines an ellipsoid with about the proportions of a pumpkin seed. The largest dimension must be less than 80 or the Atari will try to plot a point off the screen and stop with an error.

After it has drawn an outline, the program asks for "deltas for R, S, T." Give it rotational angles in degrees. To see the ellipsoid along each of its main axes, give it 90, 0, 0 the first time, -90, 90, 0 the second time, and 0, -90, 90 the third time. Then rotate it to both 45-degree positions for a rather nice, 1950s-style, atomic symbol.

DDJ

Listing 3.

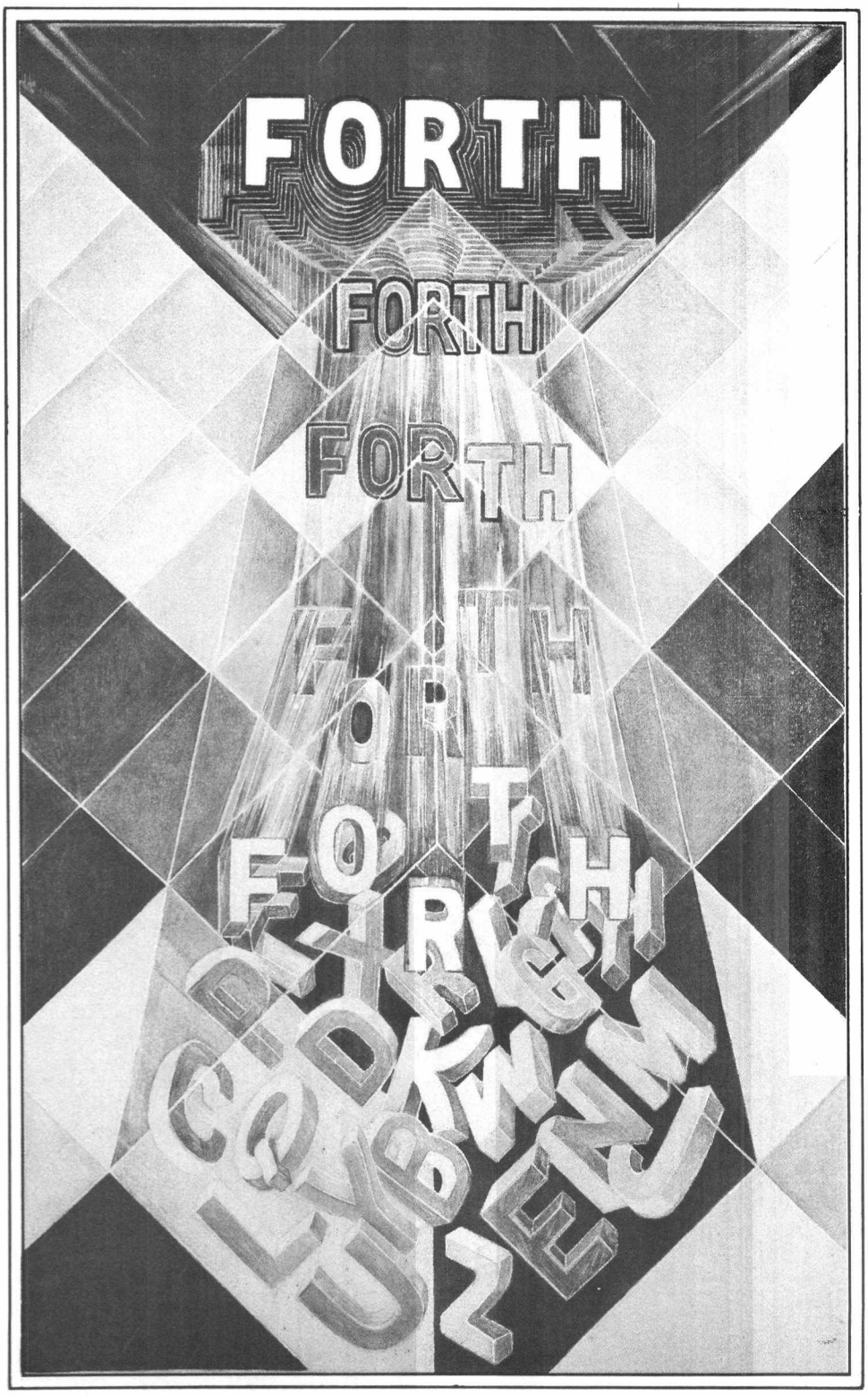
```
10 DIM SI(360):REM SINE TABLE
20 DIM CO(360):REM COSINE TABLE
30 DIM S2(360):REM SIN**2 TABLE
40 DIM C2(360):REM COS**2 TABLE
50 DIM SC(360):REM SIN*COS TABLE
70 ANGLE=5:REM DISPLAY STEP
80 REM ASK FOR SIZE OF ELLIPSOID
90 ? "X,Y,Z COEFFICIENTS";:INPUT A,B,C
100 REM
110 REM BUILD A TABLE OF TRIG VALUES
120 REM IN DEGREES, TO SPEED PROGRAM
130 REM
140 ? "BUILDING TABLES -- BE PATIENT"
160 FOR J=0 TO 360 STEP ANGLE
170 K=J*3.1415926/180
180 SI(J)=SIN(K)
190 CO(J)=COS(K)
200 S2(J)=SI(J)*SI(J)
210 C2(J)=CO(J)*CO(J)
220 SC(J)=SI(J)*CO(J)
230 NEXT J
240 REM
250 REM SET INITIAL VALUES FOR
260 REM ROTATION ABOUT X,Y,Z AXES
270 REM
280 R=0:REM X-ROTATION
290 S=0:REM Y-ROTATION
300 T=0:REM Z-ROTATION
310 REM
320 REM ATARI: CLEAR SCREEN, SET MODE
330 REM OF 320h BY 160v, WHITE DOTS
340 REM ON BLACK.  LOOP TO 360 TO
350 REM CLEAR SCREEN EACH TIME.
355 REM
360 GRAPHICS 8
370 SETCOLOR 2,0,0:COLOR 1
375 REM
380 REM THE FOLLOWING TEMPS HAVE NO
390 REM MEANING IN THEMSELVES; THEY
400 REM ARE ONLY USED FOR CALCULATION
410 REM OF THE ELLIPSE COEFFICIENTS.
420 REM *** TOP OF MAIN LOOP ***
450 K=-(SI(T)*CO(R))+CO(T)*SI(S)*SI(R)
460 L=SI(T)*SI(R)+CO(T)*SI(S)*CO(R)
470 M=CO(T)*CO(R)+SI(T)*SI(S)*SI(R)
480 N=-CO(T)*SI(R)+SI(T)*SI(S)*CO(R)
490 P=S2(S)/(A*A)+C2(S)*S2(R)/(B*B)+C2(S)*C2(R)/C*C
500 Q=-CO(T)*SC(S)/(A*A)+CO(S)*SI(R)*K/(B*B)+CO(S)*CO(R)*L/(C*C)
510 R1=-(SI(T)*SC(S)/(A*A))+CO(S)*SI(R)*M/(B*B)+CO(S)*CO(R)*N/(C*C)
520 REM
530 REM NOW CALCULATE THE VALUES OF
540 REM THE NEEDED COEFFICIENTS.
550 REM XX,YY,XY GIVE THE COEFF.S
560 REM THAT ARE MULTIPLIED BY X**2,
570 REM Y**2, AND XY FOR THE ELLIPSE.
580 REM
590 XX=C2(T)*C2(S)/(A*A)+K*K/(B*B)+L*L/(C*C)-Q*Q/P
600 YY=S2(T)*C2(S)/(A*A)+M*M/(B*B)+N*N/(C*C)-R1*R1/P
610 XY=2*(SC(T)*C2(S)/(A*A)+M*K/(B*B)+L*N/(C*C)-Q*R1/P)
630 REM
640 REM PLOT THE RESULTING ELLIPSE
650 REM THIS IS A SLOW BUT SIMPLE
660 REM METHOD -- USE PITTEWAY'S
670 REM ALGORITHM FOR ANY SPEED.
680 REM
690 RD=SQR(1/(XX*C2(0)+YY*S2(0)+XY*SC(0)))
700 PLOT RD*CO(0)+150,RD*SI(0)+80
710 FOR D=ANGLE TO 360 STEP ANGLE
720 RD=SQR(1/(XX*C2(D)+YY*S2(D)+XY*SC(D)))
730 DRAWTO RD*CO(D)+150,RD*SI(D)+80
740 NEXT D
750 REM
760 REM GET DELTAS FOR THE ANGLES,
770 REM AND DRAW THE ROTATED FIGURE.
780 REM
790 ? "DELTAS FOR R,S,T";
800 INPUT DR,DS,DT
810 DR=ANGLE*INT(DR/ANGLE):R=R+DR
820 DS=ANGLE*INT(DS/ANGLE):S=S+DS
830 DT=ANGLE*INT(DT/ANGLE):T=T+DT
900 REM KEEP ANGLES IN RANGE
910 IF R>360 THEN R=R-360
920 IF R<0 THEN R=R+360
930 IF S>360 THEN S=S-360
940 IF S<0 THEN S=S+360
950 IF T>360 THEN T=T-360
960 IF T<0 THEN T=T+360
970 GOTO 370
```

$2.50
($3.00 in Canada)

DR. DOBB'S JOURNAL

For Users of Small Computer Systems

Number 71 September 1982


FORTH
FORTH
FORTH

DR. DOBB'S JOURNAL
For Users of Small Computer Systems

September 1982 Volume 7, Issue 9

Editor – Marlin Ouverson
Assistant Editor – Fritzi Lareau
Contributing Editors –
Dave Caulkins, Dave Cortesi,
Ray Duncan, Kim Harris,
Gene Head, Michael Wiesenberg
Marketing Director – Craig Harper
Marketing Assistant – Beatrice Blatteis
Advertising Director – Laird Foshay
Circulation Manager – Gloria Romanoff
Circulation Assistants –
Theresa O'Rourke, Terri Marty
Art Director – Clifford West
Production Manager – Barbara Ruzgerian
Production Assistant – Jane A. McKean
Typesetter – Renny Wiggins
Cover Illustration – Margaret Tsai

© 1982 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $40 for first class to Canada and Mexico; $58 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Thomas H. Grohne, G. V. Elkins, Robert M. Connors, William J. McEown, Robert C. Luckey, Robert N. Stabler, DeWitt S. Brown, John B. Palmer, Burks A. Smith, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Howard E. Decker, Jim Wilson, Frank Lawyer, Chris Pettus, Paul Lutus, Ben Goldfarb, Rodney Black, Steven Weiss, Pat Phelan, John Brodie, James Fleming, Unison World, Thomas Davis, Jan Steinman, Ronald E. Johnson, G. Hunter, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor, Jr. **Sustaining Subscribers:** Pete Roberts, Steven Fisher Enterprizes. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Hofacker-Verlag, Tegernseer Strass 18, D-8150 Holzkirchen, West Germany. **Asia & Australia:** ASCII Publishing Co., Ltd, Aoyama Building 5F, 5-16-1 Minami Aoyama, Minato-ku, Tokyo 107, Japan.

CONTENTS

Articles

16 Forth Floating-Point Package
by A. J. Monroe
For those who prize portability, these Forth routines will prove valuable. But the author also shows readers who already have an assembly language floating-point package how to link that work to Forth, in what may be record time!

30 Forth Programming Style
by N. Solntseff
The morning after: can you still read the Forth code you wrote the evening before? Do you want to get your work published, but editors keep asking you what your program is supposed to *do*? Here is one person's remedy for such woes . . .

34 Relocating, Linking Loader in Forth
by Joe Barnhart
This is a fine utility for Forth programmers who are tired of waiting more than three minutes to compile an editor, for example, that can be loaded – with this software – in five seconds. It will allow you to link pre-compiled modules and load them at new addresses.

42 H-19 Screen Editor in Forth
by Albert S. Woodhull
The various hardware editing features of the H-19 make it a popular terminal for programmers. This screen editor, written in Forth, takes advantage of them in an easy-to-use way. "What I hope I have created," the author told us, "is an editor that can be used without any additional reference manuals or cards."

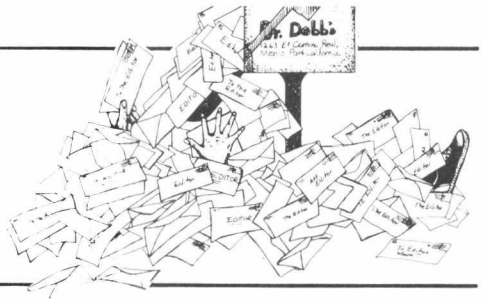
48 Z8000 Forth
by Lou Odette
This article and software will serve those looking for an example of the threaded, interpretive language, as well as those who wish to implement Forth on a Z8000 system. The latter will only have to customize two subroutines handling terminal I/O, but familiarity with Forth is assumed.

Departments

6 Letters
7 Editorial
8 Dr. Dobb's Clinic
QUITIF and Loglan
64 16-Bit Software Toolbox
8086 ASCBIN and BINASC
66 Book Reviews
Starting Forth, *Discover Forth* and *Programming Language Translation*
69 Software Review
The Working Forth Learning System
72 Of Interest

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, Box E, 1263 El Camino Real, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025 · 415/323-3111

LETTERS



Small System Interest

Dear People,

I really appreciate the articles you choose to publish, mostly oriented toward the small-system user. Mostly because of the content and character of these articles, I have gotten into "C," using Walt Bilofsky's C80 compiler. His business is The Software Toolworks, and he used to advertise in *DDJ*, but it seems that he is concentrating his advertising more in the Heath-only oriented publications now. His expansion of Ron Cain's Small-C for the Heath H8 and H89 systems handles everything but floating-point. I have modified the H9 terminal on my H8 system for program control of cursor up/down/left/right/home, erase-page and erase-to-end-of-line, and rewired some keyboard switches for single-key input of some of the most-used control characters, deciding to go that route rather than buy what I think is an over-priced H19, especially when I had the H9 already.

I was thinking about writing a screen-oriented editor in the assembler language, and had actually started some of the ground-work and coding. When the January '82 issue of *DDJ* came, I dropped the whole business and typed in the editor in C by Ed Ream. I have full-screen editing now, after adapting the editor to the C80 compiler, HDOS operating system, and special H9. It's a bit slow, since the H9 is connected through a 4800-bps serial port, and has no direct-cursor addressing; but the features of the editor, both the original features from Ed Ream and some improvements by yours truly, make my system time much more productive than when all I had was a line- and character-oriented editor.

Please keep that small-system-user interest.

Dwain Henderson
172 Country Club Dr.
Florida, NY 10921

Grist for the Mills

Dear *Dr. Dobb's*:

Byte has the ads, but you've got the content!

I'm writing in regard to Jonathan Mill's articles on the Z-88 and XASM-86, particularly Part II. I am writing to ask you to try to persuade Mr. Mills to share with us his "Daughter Board" for adding an 8087 to the minimum-mode 8088, either as an article or add-on board. Thanks, and keep up the original work.

Best wishes,
Alan D. Howard
Rt. 3, Box 680
Crozet, VA 22932

Multi-68000 Homebrew

Dear sir:

I just finished reading Al Kossow's article on the multi-68000 homebrew system that he and his friends are developing (*DDJ* #68). Please publish his 68000 assembler in Fortran. Any other 68000 articles will also be well-received by the hardware freaks like me.

I'm also developing a DEC μ/T-11 16-bit-cpu-based system with a 64 Kbyte dynamic RAM system. Any interest for this from readers?

Sincerely,
Christopher J. Shaker
1124 High St.
Hillsboro, WI 54634

Dear Editor,

Here's one note to publish the 68000 cross-assembler mentioned in *DDJ* #68.

The 68000 is very unusable without an assembler; of course, a cross-assembler written in Pascal would be better, but beggars can't be choosers.

Thanks,
Eric C. Rehnke
1067 Jadestone Lane
Corona, CA 91720

Dear *Dr. Dobb's*:

I am submitting one clamor for Al Kossow's 68000 Fortran cross-assembler. No sense in reinventing the wheel. The more public domain software, the better.

Thanks,
Milt Zlatic
9605 Carrimae Ct.
Crestwood, MO 63126

Editor's note: The above are a small sample of the many letters we received asking for the entire listing prepared by Al Kossow. You asked for it . . . next month it appears.

High-Resolution Vector Display

Dear Sirs:

As a 68000 homebrewer myself, I was very interested in Al Kossow's article in the June 1982 issue. I must take exception, however, with his statement that "there was no practicable way to ever build a high-resolution vector display capable of displaying color."

I am currently building such a display, and whilst not a trivial exercise, it is certainly practicable. Currently available high-resolution color display tubes are capable of about 1000 x 1300 pixel resolution on a 20-inch diagonal screen, so 10-bit D-A converters were chosen, giving 1024 x 1024 dots.

The key to getting good vector density without flicker is to use a constant writing speed, (i.e., the time to write a vector is proportional to its length). A very low inductance deflection yoke keeps the writing speed up and allows a full-screen width to be written in 10 microseconds. Allowing 20 ms as the limit of writing a whole screen before the onset of flicker, this allows a total length of vectors of over 700 m, surely enough for the most detailed graphics?

Keeping the writing speed constant also keeps the brightness constant and allows beam modulation to be easily applied; in fact, the 6 bits left over in each coordinate word are used to encode brightness information. A future addition will be a curve generator to generate arcs of circles from three pairs of coordinates. This will save enormous amounts of memory in drawing smooth curves and, furthermore, no point of the curve except the three given points needs to lie on the 1024 x 1024 grid.

Dynamic convergence is, of course, necessary, but given a quality deflection yoke, no terms higher than the second order should be necessary, and two analog multiplies and a handful of op-amps does the trick.

Having built many raster-scan displays, I would say that they are not as straightforward as they may seem – certainly a far cry from the domestic television which they superficially resemble. Vector refresh is a useful alternative, even for high-resolution color, and is really not that much more difficult to build. And the latter, of course, is really economical on memory. My system, allow-

ing a maximum of 8191 vectors, has 32 Kbytes of refresh memory. A 1024 x 1024 raster-scan display, 3 colors, 2 bits per color, would need 768 Kbytes. Even if memory were 1¢ a gigabyte, there still remains the problem of addressing and manipulating all this memory. Pending getting around to writing some software for the 68000, I have been testing the vector generator on a small CDP1802 system with success. I hope this letter may persuade some readers to give vector refresh a try.

Yours sincerely,
Greg Trice
1131 Sandhurst Circle #111
Scarborough, ON M1V 1V5
Canada

68xx Blurb

Dear Editor,

A users' group is forming for those interested in the 6809 processor, the OS-9 operating system, and UNIX-like systems on 68xx machines. All are welcome to participate with their ideas and other contributions. A Computer Bulletin Board System has been temporarily established to foster the group's organization and to share information. Anyone with an ASCII terminal and 300 baud modem may use this public information utility by following these instructions:

Dial 312-397-8308 (This is in a Chicago suburb.)

When a connection is established, alternately press the carriage-return key and the line-feed keys (ctrl-m and ctrl-j) until you receive the system message: PLEASE LOG IN

Respond with: HELLO-G500,,3. No spaces or other changes may be made. System instructions will follow.

Please logoff formally, not by simply hanging up. You will have an opportunity to leave suggestions or other information to the system operators.

Further information may be obtained from the bulletin board or from me. Thank you very much on behalf of all 6809 and OS-9 users.

Sincerely,
George C. Dorner
2163 Vermont St.
Rolling Meadows, IL 60008

DDJ

Editorial

Mirror, Mirror ...

When we told a friend, John Gilmore, that *DDJ* was planning to publish a Forth issue, he said that the language had always seemed a little familiar. Then, John said, he realized that it looks like LISP, but written backwards and without the parentheses. If you are new to Forth and have trouble reading the listings, try looking at them in a mirror. . . .

* * *

Everyone has his own programming style. Most languages are easily bent into, or out of, shape and can still get the job done. The ready example is, of course, BASIC, with its sly GOTOs ready to slip between any two otherwise sequential statements. The slightest afterthought or inspiration on the programmer's part means another GOTO, trailing its spaghetti-like path of logic before and after.

A data base manager we know recently spent until four o'clock in the morning fixing a payroll routine that chose to fail after several years of faithful service. The unreachable author of the code had left behind him a trail of employers as indecipherable as his program, the only consolation for which was that he problably wouldn't have been able to read the program, either. After a night of tracing and patching, the paychecks ricocheted off the printer and the manager stumbled into bed.

Most of us, of course, pride ourselves on the legibility and cleverness of our programs. We may stop short of taping our favorite work to the hall mirror, but there is a certain amount of pride in a job well done. Until someone else tries to read the listing. Even then, we tend to believe that the other person simply doesn't possess the acumen to readily understand what our code does, and that its elegance baffles them. If we must make concessions, they are usually in the form of comment lines, which should be used with clean code, not as a substitute for it.

Forth has been called a cryptic language; others say it is more hieroglyphic. The difference of opinion may result from personal preference, or from studying programs written in different styles. (Given the chance, would you really use long, distinct variable names? Or would you revert to encrypted forms requiring a look-up table to comprehend?) Any language can be made to seem opaque, intentionally or otherwise. Most languages, including Forth, can also be made to display order, logic and consistency to all but the most untrained of eyes. Perhaps by comparing the listings in this issue, you will weigh the importance of style and approach, as well as the results.

* * *

Some people think Forth has outlived its practical usefulness. It should serve, they say, as a learning tool for extensible interpreter/compiler models of the future. But there are several thousand voices, and more daily, testifying to Forth's flexibility and power. One thing is certain: Forth is a language in transition. Not all members of the standards committee left the 1979 meeting in agreement. As a result, subsequent meetings have occurred in order to bring about a consensus. We can look forward to something which, depending on its release date, will probably be called Forth-83.

– Marlin Ouverson

DR. DOBB'S CLINIC

by D.E. Cortesi

Conditional Submit

In the June Clinic we asked if someone wouldn't like to write CP/M commands to abort a submitted job under different circumstances – if a submit operand was omitted, or if it was an ambiguous file reference, etc. Don Wright of San Antonio, TX, responded that he would, and he did. He called his program IF, but when we typed it in we renamed it QUITIF since that seemed to express its function more clearly.

The QUITIF program (Listing 1, page 14) takes two parameters. The first is a keyword, one of AMBIG, EXISTS, MISSING, or NULL (actually, it looks only at the first letter of the keyword). The second parameter would usually be, in whole or in part, a SUBMIT parameter. That parameter is tested in one of four ways depending on the keyword; then QUITIF either kills the submitted job or lets it continue.

Here are some examples. In a submitted file, the line

```
QUITIF AMBIG $1
```

will cause the submit file to be aborted if the first parameter of the submit was an ambiguous file reference. The line

```
QUITIF EXISTS $2
```

will abort the job if the second parameter of the submit named a file that existed. (If that parameter were an ambiguous reference, the job would be aborted if there were *any* file that matched the parameter.) The command

```
QUITIF MISSING $3
```

is the inverse; it will abort the job if there is no file that matches the given reference. Finally, the NULL keyword tests the second parameter itself, not the file it refers to. If the second parameter is completely absent, the job is ended.

Note that you have quite a bit of freedom in forming the second parameter. For example,

```
QUITIF MISSING B:$1.ASM
```

might be appropriate in a submit file for doing assemblies.

Prophecy Fulfilled

We quote with great pleasure this passage from the version 1.1 manual for IBM PC BASIC (page I-11):

"When BASIC wants to branch to a particular line number, it . . . has to search through . . . the program, starting at the beginning, to find the line it's looking for.

"In some other BASICs this search must be performed each time the branch occurs . . . . In IBM Personal Computer BASIC, the search is only performed once, and thereafter the branch is direct. *So placing frequently-used subroutines at the beginning of the program will not make your program run faster.*" [emphasis added]

Loglan Lives

Little fixes and improvements to existing systems are all very well, but we at *DDJ* like to think of our readers as hardy pioneers, working 'way out on the frontier of personal computing. In that spirit, we'd like to tell you about something that lies just over the next mountain range, and how to find the blazes that mark the trail toward it.

In the summer of 1965, a friend gave me a knee-high stack of back issues of *Scientific American*, and I read them all. Most of the articles I read then faded from memory long ago, but one of them stuck to my imagination like a leech. It was entitled "Loglan," appeared in the June, 1960, issue and was written by James Cooke Brown, then at the University of Florida.

Brown described the design and construction of the artificial language Loglan and the uses to which it could be put. The design of Loglan had a marvellous elegance. The language and its uses were endlessly fascinating to anyone who liked verbal play, logical puzzles, or grand philosophical inquiries. At the end of the 1960 article, Brown promised that a Loglan primer and teaching materials would soon be published. In 1965, a careful search of the reference sources of the San Francisco Public Library turned up absolutely nothing further on J. C. Brown or on the Loglan project. The language had, it seemed, dropped into limbo.

Twice during the following decade, the memory of Loglan returned to haunt me with sufficient force that I reread the old article and tried again to locate some sign of Loglan in print, but it remained invisible. Then, in 1976, a tiny ad appeared in an issue of *SciAm*. Something called The Loglan Institute had been founded, offered a primer and a dictionary, and was publishing *The Loglanist*, a journal of Loglan studies! I responded at once, and have been an interested observer of the progress of the language ever since.

Until recently I felt that Loglan would not interest readers of this column. It's true that most of you (according to our reader surveys) have a strong interest in programming languages, and so are probably the kind of people who would find the construction of a new *human* language a fascinating project. Furthermore, from its birth, Loglan has carried a strong promise of being useful as a computer interface language. However, Loglan has been in a state of flux and expansion for some time; the papers in *The Loglanist* have been painfully detailed and hard to read; and there was no practical relevance to personal computing. Now, at last, the situation is changing. In particular, a major piece of work has been completed that makes Loglan relevant, and Loglan experiments accessible, to many computer enthusiasts.

The Genesis of Loglan

Loglan was first conceived as an experimental tool to test the "Sapir-Whorf Hypothesis." Crudely put, the hypothesis proposes that our minds can only comprehend those things that our language can express. The structure of language, in other words, *is* the structure of thought, so the language that we use determines (in some sense) the boundaries of what we can think. It's a fairly broad hypothesis. Since it crosses so many disciplines (psychology, sociology, linguistics, to name a few), and since any measure of "what we can think" would be confused by an immense number of variables other than language, it is generally assumed that the hypothesis can't be tested in any scientific way. You can argue the matter from any of a thousand points of departure (programmers would start with the common observation that a change of programming language – from BASIC to Forth, say – can release a whole new class of problem solutions to view), but an experimental test would seem to be impossible.

Or would it? Brown reasoned that if one devised a new language, one that was in some way radically different from all other human languages, and taught that language to a large sample of people, then some measurable difference might show up between the way those subjects thought in their native language and in the new one. The new language would have to be a usable, speakable, human

language; it would have to be at least no harder to learn than a natural language; and yet it would have to be unlike any other language. Loglan was the result of his efforts.

The Shape of Loglan

The original *Scientific American* article is still a good summary of the structure of Loglan. A number of details have changed since then, but if you want to learn more about Loglan, your best first step is to visit your local public library and look up that article. Here I will point out a few of the design features that relate Loglan to computing.

The most striking thing about Loglan is the orderliness and regularity of its design. It is like the better computing languages (e.g. LISP, Forth) in that it is composed of a very few parts which are connected using a very few general rules. Words are built up from an alphabet of just twenty-two letters: five vowels and seventeen consonants (you will count only sixteen in the 1960 article; *h* was added in 1980). Each letter represents just one sound; the English ambiguity of "there," "they're," and "their" cannot occur. Punctuation is represented by spoken words; the ambiguity of "its" and "it's" can't happen.

There are only three classes of words ("parts of speech"): predicates (which play the combined roles of English verbs, nouns, and adjectives), structure words (connectives, punctuation, and "variables" which are, roughly, pronouns), and proper names. Each class is distinguished by its pattern of consonants and vowels; a simple algorithm would allow a computer to identify the class of any word. Finally, the language as a whole is unambiguous. There can be no doubt as to the role of any word in an utterance. In Loglan, time flies like an arrow *precisely* the way fruit flies like a banana; there isn't any other way of saying it. It is possible to be ridiculous or metaphorical in Loglan, but it is not possible to be ambiguous.

The Development of Loglan

From 1955 when he began work, until 1976 when he set up The Loglan Institute, James Brown had Loglan pretty much to himself. I picture him working alone, much in the style of that other obsessive linguiphile, J. R. R. Tolkein, although without Tolkein's freedom to invent a universe to contain his language.

The Institute's ads attracted a membership that has remained constant around 200 subscribers since 1976. The addition of so many minds to the development of the language produced a great deal of activity. The syntax was extended and improved in a number of ways, and many new words were added. An important milepost was passed in 1977, when Brown and two other people were able to engage in several hours of free, spontaneous discourse completely in Loglan. Twenty-two years after its invention, Loglan had been demonstrated to be a speakable, learnable, human tongue. There may be as many as a dozen fluent Loglan speakers in the world today, and perhaps fifty more who can read and write it with some ease (I am not one of them).

A major concern of the Institute has been the achievement of unambiguity. The Loglan primer (*Loglan 1*, J. C. Brown, The Loglan Institute, 1975; now out of print) contained as an example the possible meanings of the phrase "pretty little girl's school," and showed how each would have to be rendered in a unique Loglan phrase. A long correspondence on the "PLGS problem" exposed some shortcomings in the syntax used to group modifiers, and led to some deep changes in the language.

The problem remained, how could Loglan be *proven* to be unambiguous? Small bugs (time flies?) kept cropping up unexpectedly. This was important, for the language was coming to be seen more as a way of communicating with computers and less as an experimental tool in linguistics, and proven ambiguity was significant to that use. The solution, and a proof, came with the presentation of the grammar of Loglan as a deterministic, finite rule-set that can be parsed with standard compiler techniques. And that brings us to the work at hand.

The Machine Grammar of Loglan

The latest output of the Loglan Institute is a machine grammar of Loglan; specifically, the input to the well-known YACC utility of UNIX that will cause YACC to generate a Loglan parser. The machine grammar, or MACGRAM, package is available as a 173-page xeroxed paper. It contains four parts. The first is the YACC input: a set of BNF-like rules, each clearly annotated. The second part is a lengthy glossary of the terms used in the grammar rules and in the notes on them. The third part is a set of some 700 Loglan sentences and fragments which are the test database for a parser, with their English translations. The final part is the source code, in C, for a pre-parser, a module that tokenizes Loglan text for presentation to the parser.

MACGRAM is a very significant achievement (its contributors include Sheldon Linker, Jeff Prothero, Scott Layson, Doug Landauer, and Bob McIvor, and of course Brown himself). Loglan has been shown to be a genuine language; MACGRAM is therefore the first-ever complete machine parser for a human tongue. Nothing like it has ever been done before. It may be more significant to readers of *DDJ* that some of the most recent work has been performed on a Heath H89, to which McIvor has managed to transport a version of YACC!

Loglan, the Future and You

What is the future of Loglan in computing, or in any other endeavor? No one can say, yet. Your intern is prepared to bet that it *has* a future, however. It is a unique language for computing. Computer output of Loglan speech is trivial; the ASCII characters of Loglan text map directly onto Votrax phoneme codes without any complicated text-to-speech algorithm. The simple phoneme structure, and the one-for-one mapping of sounds to letters, means that Loglan has to be by far the simplest language for machine recognition, whether of speech sounds through a microphone or of letters through an optical reader.

A useful dictionary of "basic Loglan" would be only a very modest disk file. Loglan syntax can be parsed easily and without ambiguity. A system that takes Loglan input, "understands" it, acts on it, and reports its output in Loglan speech, *all in real time*, could conceivably be built on no larger a hardware base than a typical CP/M-80 system!

But not yet. Loglan is not quite ready to "go public" again. Brown and the members of the Institute are in the midst of a "Great Morphological Revision," in which a significant part of the Loglan vocabulary is to be rebuilt for better recognizability and easier learning. Also, there is little in the way of up-to-date tutorial material. The 1975 primer is out of print, and both it and the 1975

dictionary have been outdated by work since. There is a great deal of work to be done to make a presentable, finished, Loglan learning package.

If this account has intrigued you, start by reading the 1960 article. If the elegance of the language then catches your mind the way it caught mine, you will want to participate in, or at least keep current on, the development of the language. You can do that by joining the Loglan Institute. Your involvement can be at any of several levels.

A subscription to *The Loglanist* costs $20 for approximately four issues (the journal is published occasionally, and the cost of each issue is charged to your current balance.) Send an extra $20 and request copies of "the Primer section of the June 1978 distribution" (a draft of the first chapters of a new primer), of Volume Four Number Two (a summary of language changes 1975-1980), and of the MACGRAM paper. Or, for biennial dues of $100, you can be a regular member of the Institute, receive a regular newsletter, and have a voice in the affairs of the Institute. Then, if you are *really* hooked, you can inquire into spending your vacation at the Institute, working on the language.

The address is The Loglan Institute, Inc., 2261 Soledad Rancho Road, San Diego, CA 92109. Welcome to Loglandia!

(QUITIF Listing begins on page 14)

QUITIF Listing

To abort submitted jobs, IF . . .

(Text begins on page 8)

```
; QUITIF -- a submit utility for conditional aborts
;
;Usage: QUITIF AMBIG $1    -- abort on ambiguous name
;       QUITIF EXISTS $1   -- abort on pre-existing file
;       QUITIF MISSING $1  -- abort on non-existent file
;       QUITIF NULL $1     -- abort on null parameter
;
; Only the first letter of the keyword is checked.  A letter
; other than A, E, M, or N (or an omitted keyword) will cause
; an unconditional abort.
;
TPA     EQU     0100H
BDOS    EQU     0005H
FCB1    EQU     005CH   ; the default FCB in low storage
NAME1   EQU     005DH   ; filename number 1 (keyword)
FCB2    EQU     006CH   ; second parameter, treated as FCB
NAME2   EQU     006DH   ; filename number 2 (argument)
;
PRTSTR  EQU     9       ; print string until $
SEARCH1 EQU     17      ; search for first matching name
ERASE   EQU     19      ; erase file
;
XIF     ORG     TPA
        LXI     H,NAME1
        MOV     A,M     ; get function code letter
        LXI     H,NAME2 ; point to filename/parameter
;
        CPI     'A'     ; QUITIF AMBIG name
        JZ      AMBIG
        CPI     'E'     ; QUITIF EXISTS name
        JZ      EXISTS
        CPI     'M'     ; QUITIF MISSING name
        JZ      MISSING
        CPI     'N'     ; IF NULL name
        JZ      NULL


;
; keyword is none of the above, do unconditional abort
;
KILL    MVI     C,ERASE ; abort the submit stream
        LXI     D,SUBFILE
        CALL    BDOS    ; ..by erasing $$$.SUB
        MVI     C,PRTSTR
        LXI     D,KILLMSG
        CALL    BDOS    ; ..and tell about it.
        RET             ; exit to CCP


;
; abort if the second parameter is an ambiguous name
;
AMBIG   MVI     A,'?'
        CMP     M
        JZ      KILL    ; it is one -- abort
        INR     L
        MOV     A,L
        CPI     NAME2+11
        JC      AMBIG   ; continue to end of filename
        RET             ; unambiguous -- return to CCP


;
; abort if the file named exists
;
EXISTS  CALL    SEARCH  ; look for file
        CPI     0FFH    ; was there one?
        RZ              ;   no -- return to CCP
        JMP     KILL    ;   yes -- abort submit file


;
; abort if the named file does not exist
;
MISSING CALL    SEARCH  ; look for the file
        CPI     0FFH    ; was there one?
        RNZ             ;   yes -- return to CCP
        JMP     KILL    ;   no -- abort submit file


;
; abort if the second parameter was null, i.e. all-blank
;
NULL    MVI     A,' '
        CMP     M       ; test filename
        RNZ             ; ..not null, return to CCP
        LXI     H,NAME2+8
        CMP     M       ; test file extension
        RNZ             ; ..not null
        LDA     FCB2    ; test for a drivecode alone
        ORA     A
        RNZ             ; .."d:" given alone
        JMP     KILL    ; drive, name, extension, all null


;
; subroutine to use the search-for-first-match function
;
SEARCH  MVI     C,SEARCH1
        LXI     D,FCB2
        CALL    BDOS    ; find first matching name
        RET             ; return A=ff, 00, 01, 02, 03
;
KILLMSG DB      'SUBMIT ABORTED',13,10,'$'
SUBFILE DB      1,'$$$     SUB',0,0,0,0
;               rest of FCB is irrelevant
        END     XIF
```

End Listing

Forth

Floating-Point Package

A floating-point arithmetic package for Forth can be mechanized in two quite different ways. There are advantages to either approach. We shall briefly discuss both alternatives and then present one of them in detail. The first approach emphasizes the use of machine language routines. The second uses only two simple machine language routines, with the balance of the package written in high-level Forth.

The first approach has several advantages. Variable precision is easy to achieve; only three screens (approximately 600 bytes) of Forth programming is required, thus it loads quickly; and it runs somewhat faster than the high-level Forth version.

One disadvantage of this approach is that approximately 2.4K bytes of object code must be developed by assembly-language routines. This proves to be a non-trivial task. On the other hand, if you have already written such a package for other purposes, you can link it to Forth with less than a single evening's effort.

I think the most significant disadvantage of this approach is its lack of portability. Nevertheless, I originally took this approach because I had on hand a package of floating-point routines written in the winter of 1975 and I could reassemble them to use otherwise wasted memory. (8K bytes below the NorthStar DOS.)

The second approach has the distinct advantage that it is highly portable and requires only two simple CODE word routines to make it work.

Two disadvantages which accrue from this approach are that it generates somewhat more code (3200+ bytes, 19 Forth screens) and its precision is limited to 32 bits. The latter limitation is, of course, a direct result of the approach. Additional precision can be achieved by a rewrite, but such work seems unwarranted by any application I have encountered to date.

Another slight disadvantage is that the compiled package cannot be moved to arbitrary locations in memory, as is the case for the assembly language approach (but the source code may be compiled anywhere).

Finally, there is a minor disadvantage in that it requires a considerable amount of time to load 19 screens of Forth. I have made this package a permanent part of one of my copies of Forth in order to avoid this annoyance.

The Assembly Language Approach

Figure One illustrates this approach pictorially. We presume the existence of a set of machine-language floating-point subroutines which include a keyboard input routine (FINP) that accepts ASCII input characters, converting them to decimal binary and storing them in a known location. On exit from this routine, the decimal binary number is converted to a floating-point binary number, i.e., it is "normalized" by a call to FPNORM. Floating-point output is achieved by a call to FPOUT which essentially reverses the process of FPINP. Multiplication, addition, subtraction, and division are accomplished by calls to FPMULT, FPADD, FPSUB, & FPDIV, respectively. To illustrate the approach, consider the implementation of a Forth floating-point multiplication command F*. It can be written as a CODE word linkage to a jump table preceding the floating-point routines, e.g.,

```
CODE F* 0006 CALL   NEXT JMP
```

As I have mechanized it, the jump table resides at address HEX 0000 and up, i.e.

```
0000 ( CODE FOR JMP EXEC1 )    ; FPINP
0003 ( CODE FOR JMP EXEC2 )    ; FPOUT
0006 ( CODE FOR JMP EXEC3 )    ; FPMULT
```

where EXEC1, EXEC2, EXEC3, . . . first call a routine to save the Forth instruction pointer, then a routine to POP the required number of floating-point numbers to working locations, then call the appropriate machine-language routine, and finally jump to a routine that restores the Forth instruction pointer, pushes the results onto the stack and returns to the Forth calling address which then executes a NEXT JMP.

Typical 8080 assembly language routines which accomplish these tasks are as follows:

```
EXEC3:  CALL EXEC       ; SAVE THE FORTH IP & RET ADDR
        CALL XFR2       ; POP 2 NO.'S TO WORK LOCATION
        CALL FPMULT     ; DO A FP MULT
        JMP SETS        ; RESTR IP, PUSH RESULT TO STK
EXEC:   POP D           ; LOCAL RET ADDR
        POP H           ; FORTH RET ADDR ...
        SHLD RETADDR    ; SAVE IT
        MOV L,C         ; SAVE IP
        MOV H,B
        SHLD IP
        XCHG
        PCHL            ; RET TO LOCAL CALL
XFR2:   POP D           ; LOCAL CALLING ADDR
        POP H           ; FP EXPONENT ...
        MOV A,L         ; USES ONLY 1 BYTE
        STA EXPA        ; STR IN FPACC EXPONENT
        POP H           ; MSW OF FP NO. ....
        SHLD LSWA+2     ; STRD IN MSW OF FPACC
        POP H           ; LSW OF FP NO. ...
        SHLD LSWA       ; STRD IN LSW OF FPACC
        POP H           ; FP EXPONENT ...
        MOV A,L         ; USES ONLY 1 BYTE ...
        STA EXPP        ; STRD IN FPOP EXPONENT
        POP H           ; MSW OF FP NO. ..
        SHLD LSWP+2     ; STRD IN MSW OF FPOP
        POP H           ; LSW OF FP NO. ...
        SHLD LSWP       ; STRD IN LSW OF FPOP
        XCHG            ; LOCAL CALLING RET ADDR ...
        PCHL            ; TO PROGRAM CNTR.
SETS:   LHLD IP         ; RESTORE IP TO ...
        PUSH H
        POP B           ; ... THE BC REGISTER PAIR.
        LHLD LSWA       ; FPACC HAS RESULTS ...
        PUSH H          ; ... WHICH ....
        LHLD LSWA+2     ; ... GET ....
        PUSH H          ; ... PUSHED ...
        LHLD EXPA       ; ... ONTO THE ...
        XRA A           ; ... STACK WITH HI ....
        MOV H,A         ; ... BYTE OF EXP SET ....
        PUSH H          ; ... TO ZERO
        LHLD RETADDR    ; FORTH RETURN ADDR ...
        PCHL            ; TO THE PROGRAM CNTR
IP:     DS 2
RETADDR: DS 2
```

The jump table is not an essential requirement, but it is a convenient way to make the CODE word linkage insensitive to alterations of the assembly language package as occurs, for example, when the precision of the arithmetic is changed.

Finally, the floating-point routines are autoloaded so that their existence is transparent to the user. In my case, this was accomplished by a CODE word DCOM which links to the

by A.J. Monroe

Alfred J. Monroe, 3769 Grandview Blvd., Los Angeles, CA 90066.

disk read/write routine of the same name in NorthStar DOS. This linkage was made a permanent feature of the three Forth floating-point screens, so that it is not necessary to remember to load the machine-language subroutines when these screens are loaded.

And that's all there is to it.

As indicated earlier, it took only several hours to complete the linkage just as described. (I have, of course, completely forgotten the original agony of writing and debugging that assembly language package in 1975.)

Command Glossary

Both versions of the floating-point package support the following commands. In semi-figForth style:

F* FP#2 FP#1 ... FP#2*FP#1
Multiplies two top of stack FP#'s and leaves result on the stack.

F+ FP#2 FP#1 ... FP#2+FP#1
Adds two top stack FP#'s and leaves result on the stack.

F- FP#2 FP#1 ... FP#2-FP#1
Subtracts top of stack FP# from second FP# on the stack. Leaves result on the stack.

F/ FP#2 FP#1 ... FP#2/FP#1
Divides the second FP# on the stack by the top of stack. Leaves result on the stack.

FDUP FP#1 ... FP#1 FP#1
Duplicates the top of stack FP#.

F2DUP FP#2 FP#1 ... FP#2 FP#1 FP#2 FP#1
Duplicates the two top FP#'s.

FSWAP FP#2 FP#1 ... FP#1 FP#2
Swaps the two top of stack FP#'s.

FDROP FP#1 ...
Drop the top of stack FP#.

F! FP#1 N1 ...
Store FP#1 to address N1.

F@ N1 ... FP#1
Fetch FP# from address N1.

FVARIABLE
FP#1 FVARIABLE FNAME (DEFINING)
FNAME ... N1 (EXECUTION)
Defines a floating-point variable FNAME with an initial value FP#1. At execution time returns the address of FNAME to the top of stack.

FCONSTANT
FP#1 FCONSTANT FNAME (DEFINING)
FNAME ... FP#1 (EXECUTION)
Defines a constant FNAME whose value is FP#1. At execution time, invoking FNAME returns FP#1 to top of stack.

FVAL FP#1 FVAL FNAME (DEFINING)
FNAME ... FP#1 (EXECUTION)
FP#1 TO FNAME ... (EXECUTION)
FNAME1 TO FNAME2 ... (EXECUTION)
Defines a floating-point variable with initial value FP#1. At execution time, invoking the variable name returns its value, not its address to the stack. If the name is invoked following "TO" the FP# on the top of the stack will be stored in the variable. See *Forth Dimensions*, Vol. 1, #4, pg. 38, "The "TO" Solution," P. Bartholdi.

D>F d1 ... FP#1
Converts the double word integer on the top of the stack to floating point format, e.g.

1.0 D>F produces 10 in floating format
10. D>F produces 10 in floating format
.10 D>F produces 10 in floating format
0.10 D>F produces 10 in floating format

Note that the placement of the decimal point is irrelevant; that leading zeroes are ignored; and that trailing zeroes are not ignored. A scaled FP#, say 1.234, can be formed by:

1234. D>F 1000. D>F F/

FINP ... FP#1
Invoking FINP causes the subsequent keyboard input stream to be interpreted as a floating-point number. Error detection and recovery are provided. Cntrl-c will abort the command. Input may be either integer or "E" format. Decimal points are not allowed; they must be accounted for in the exponent, i.e.

-1.234 is input as -1234E-3

"+" signs are not permitted. Input is terminated by a space. FINP may be used within Forth words. A maximum of nine digits should be entered. (Also see D>F.)

F. FP#1 ...
This is the floating-point output routine. The output number format is:

sign, digit, decimal point, 8 digits, space,
exp, space, n1

For example, 12.34 in FP format will be output as:

+1.23400000 EXP 1

All of the above commands may be executed in the immediate mode or included within other Forth words.

All arithmetic operations are rounded to 32 bits of precision in the case of the high-level Forth version. In the case of the machine language version, precision is set at assembly time in multiples of 8 bits.

High-Level Forth Approach

The second approach is illustrated pictorially in Figure Two. There are 17 screens of high-level Forth supported by two essential CODE words for a total of 19 screens.

The format of a floating-point number on the stack is:

d1 n1 ...

where n1 is a signed word representing the exponent and d1 is a signed double word representing the mantissa.

The two essential CODE words (screens 90 and 91) are <-S and S->. Code word <-S performs a logical left shift on a double word. Code word S-> performs a logical right shift on a double word. Both expect the top word on the stack to be a logical carry. This word is popped and tested. If it is one, <-S shifts a one into the least significant bit of a double word on the stack; S-> shifts a one into the most significant bit of a double word on the stack. Both return a word to the top of the stack which indicates the state of the carry flag at the conclusion of the shift operation. In effect, these words perform a multiplication and division of a positive double word by two. It will be noted that they are never used on negative double words in the routines, although this is a somewhat arbitrary restriction. In fig-Forth glossary format, for positive double words:

```
<-S   d1  cy0 ... d1*2 cy1
S->   d1  cy0 ... d1/2 cy1
```

Figure Three shows the binary format chosen to represent a floating-point number. All numbers are represented in two's complement form, i.e. the high-order bit is zero if the number is positive, and one if the number is negative. We assume a "binary point" between bit 31 and bit 30. This is equivalent to stating that all numbers are stored in the range plus one to minus one. That is, bit 30 has a binary weight of 1/2, bit 29 has a binary weight of 1/4, and so on.

The exponent will be held as a signed, most-significant word and it will represent the power of two by which the mantissa is to be multiplied to obtain the correct number.

To input a number in floating-point format, and to rectify floating-point numbers after an arithmetic operation, we require a normalization routine. In fact, we require two; DNORM in screen 92 and DDNORM in screen 96. DNORM will be used on double words that result from addition, subtraction, and division. DDNORM will be used on quad-word mantissas which result from a floating-point multiplication.

The normalization process works as follows. Consider a one in bit 0. In Forth, this is the positive integer "one." If we shift this bit left until it is in bit 30, we will have made a total of 30 shifts and the value represented will have a binary weight of 1/2. Hence we need a binary exponent of one because

$$2^1 * 1/2 = 1$$

Thus if we set our binary exponent initially to 31 and subtract one from it for every logical shift left, we will end up with the integer one properly represented in floating-point format. A little thought should convince the reader that all integers (up to our allowed maximum of nine) will be properly converted to floating point format by this procedure.

The word D>F simply uses DNORM and an exponent of 31 to achieve the floating-point representation of a double-word integer, e.g. unity in floating-point format can be accomplished by keyboard input of 1. D>F.

Screens 95, 96, and 97 develop the floating-point multiplication routine. The key words required are <-S, S->, and D*. The latter word deserves some explanation. It performs a multiplication of two positive double words using the following artifice. We note that a positive double word d1 is made up of two consecutive single words:

```
        d1 ...
    n2  n1 ...
```

d1 can therefore be quite correctly represented as the sum of two double words:

```
n2  0  0  n1  D+  --->  d1
```

Letting LO1 represent n1, and HI1 represent n2, the multiplication of one double word by another becomes

$$(LO1 + 2\uparrow16 * HI1) * (LO2 + 2\uparrow16 * HI2) = LO1*LO2 + HI1*LO2*2\uparrow16 + HI2*LO1*2\uparrow16 + HI1*HI2*2\uparrow32$$

If LO1*LO2 is formed using the unsigned double-word multiply U*, then the least significant word of the double-word result will be the partial product representing the least significant word of a four-word product, and the most significant word of that double-word product, will be a carry which is to be added to the least significant word of the double-word partial product formed by HI1*LO2 using U*. HI2*LO1 formed by U* must be added to this partial result. The least significant word of this triple sum is then the next-to-least-significant word of the four-word product, and the most significant word is a carry to the least significant word of the double-word product of HI1*HI2.

If all this sounds confusing, try it in decimal,

$$1234. * 5678. = 34*78*10\uparrow0 + 34*56*10\uparrow2 + 12*78*10\uparrow2 + 12*56*10\uparrow4$$

```
12*56*10↑4 =   6  72  00  00  = HI2*HI1*10↑4
34*56*10↑2 =      19  04  00  = LO2*HI1*10↑2
12*78*10↑2 =       9  36  00  = HI2*LO1*10↑2
34*78*10↑0 =          26  52  = LO2*LO1*10↑0
             ---------------
             07  00  66  52
```

Note that products of any degree of precision can be formed by this cascading multiply and sum-the-carry process. Also note that D* works with *positive* numbers only. This is not a limitation, of course, since we will apply the word DMINUS to any negative mantissa before carrying out the cascading process just described, and then apply DMINUS once again if the signs of the two floating-point numbers are different.

Two additional minor points should be made concerning F*. Because the binary weight of bit 30 is 1/2, the unsigned product of 1*1 in floating-point format will yield 1/4. Thus we need to compensate the exponent of the product by adding unity to it.

Further, to maintain accuracy we will do a round-off based on the third word of the normalized product. This round-off can lead to overflow into the sign bit. Accordingly, after the round-off addition is performed, the sign of the most significant word will be checked, and if it is negative, a right shift will be performed and one will be added to the exponent.

There are at least three ways to implement division. The division routine given in screens 98 and 99 is based upon the age-old shift and trial subtract algorithm. This is not particularly elegant, but it is easy to mechanize. Initially, the quotient is made a double-word zero as the third number on the stack. The dividend is made positive (if it is negative), and the divisor is made negative (if it is positive). A trial subtraction of divisor from dividend is then made. If the result is negative, a zero is

right shifted into the quotient, and the divisor is added back into the dividend which is then shifted left. If the result of the trial subtraction is positive, a one is right shifted into the quotient and the dividend is shifted left. This process is repeated 31 times.

The CASE construct used in screens 98 and 99, and elsewhere, is shown in screens 144-145. This CASE construct is due to Dr. Eaker. It is described in detail in *Forth Dimensions*, Vol. 2, No. 3, pg. 37-40. The <OF and >OF and RNG-OF constructs are augmentations due to this writer. Discussion of these constructs is beyond the scope of this article. The reader is referred to Dr. Eaker's excellent article for details.

The floating-point keyboard input routine (FINP) is given in screens 100-103. It is designed to detect certain operator input errors; provide for error recovery; and permit command abort (Cntrl-c). It can be vastly simplified if the Forth constructs WORD and NUMBER are utilized, but at the sacrifice the error recovery and operator-interactive features. The routine first generates a double-word integer on the stack which is then converted to floating-point format using D>F. If an exponent has been entered by the operator, the result of the D>F operation is then multiplied by either 10 (positive exponent) or 0.1 (negative exponent) a number of times equal to the value of the exponent. That is to say, the exponent of the normalized floating-point number is saved as a power of two, not ten.

Floating-point addition and subtraction are developed in screens 104 and 105 based solely upon the existing Forth word D+, DMINUS, and the new words S-> and DNORM. Note that adding two positive, normalized mantissas gives a guaranteed overflow into the sign bit which must be compensated for by S->.

Screens 106 and 107 develop the floating-point output routine. The scheme used is essentially a reversal of the input routine. The mantissa is first made positive. The exponent, if positive, is forced negative or to zero by successive multiplications by 0.1. If it is already negative, it is forced to zero or a positive value by successive multiplications by 10, and then it is forced negative. On exit from this procedure, the exponent will have been forced to lie in the range 0 to -4. It is then forced to zero (if necessary) by successive right shifts of the mantissa, i.e. divide by 2 a number of times equal to the residual exponent. The routine D* is then used to "spill" the mantissa one decimal digit at a time by successive multiplications by 10.

Finally, screens 109-117 present several useful functions, to wit:

FINT Returns the integer portion of a floating-point number in floating-point format.

FMOD Returns the remainder of a floating-point division in floating-point format.

SINE Returns the sine of an angle expressed in degrees. Error is of the order 2*10↑-9.

COSINE Returns the cosine of an angle expressed in degrees.

FSQR Returns the square root of a number.

ARCSIN-DEG Returns the arcsin in degrees. Error is of the order 2*10↑-8.

ARCSIN-RAD Returns the arcsin in radians.

The sine and arcsin routines are based upon polynomial approximations given in section 4 of the *Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables*, U.S. Dept. of Commerce, National Bureau of Standards, Applied Mathematics Series #55, June, 1964. Although I have not required logarithmic or exponential functions in my applications so far, they can be just as readily written in Forth using the polynomial approximations given in the above reference.

Conclusion

The Forth package described here was written out of curiosity. As stated at the outset, this writer has always preferred the assembly language approach to the implementation of floating-point routines. I must confess, however, that my assembly-language routines are considerably less readable than are the Forth routines, and since they consist of some 1100+ essentially undocumented statements, there is much to be said for writing the package in high-level Forth; especially when it comes to modifications – or just plain debugging.

This writer is thoroughly impressed with Forth. It took

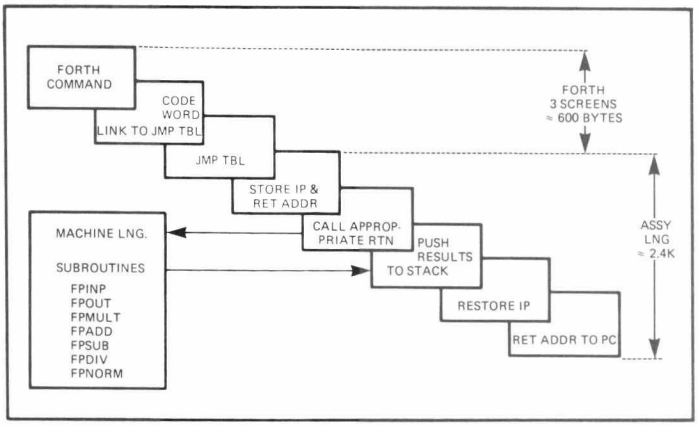


Figure 1.
Assembly language approach to floating-point package.

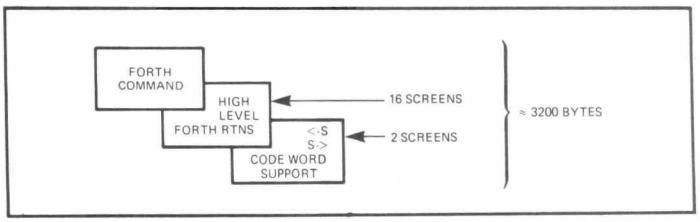


Figure 2.
High-level Forth approach to floating-point package.

(Continued on page 71)

Floating-Point Program
(fig-Forth)

(Text begins on page 16)

Editor's note: Due to limited space, blank lines have been deleted from the normally fifteen-line screens which follow. Screens #144 and #145 are CASE construct augmentations previously published in *Forth Dimensions*, Volume III, No. 6.

```
SCR # 88
  0 ( FORMATS FOR FLOATING POINT PACKAGE )
  1 ( MANTISSA IS FOUR BYTES, EXPONENT IS 2 BYTES. BASED ON )
  2 ( D FORMAT" WHICH IS - )
  3 ( LSW MSW ..... )
  4 ( FLOATING POINT FORMAT IS - )
  5 ( LSW MSW EXP .....  )
  6 ( FLOATING POINT STORAGE IS - )
  7 ( LO ADDR       HI ADDR )
  8 (  EXP     MSW     LSW   )
  9 ( LB-HB    LB-HB   LB-HB )
 10
 11
 12 -->

SCR # 89
  0 ( FLT PKG - DEF OF FVARIABLE, FCONSTANT, F!, F@, F0, CNSTS )
  1 ( FDROP, NDROP )
  2 : NDROP 0 DO DROP LOOP ;
  3 : FDROP 3 NDROP ;
  4 : F! DUP >R 2! R> 4 + ! ;
  5 : F@ DUP >R 4 + @ R> 2@ ;
  6 : FVARIABLE <BUILDS HERE 6 ALLOT F! DOES> ;
  7 : FCONSTANT <BUILDS HERE 6 ALLOT F! DOES> F@ ;
  8 0 0 0 FCONSTANT F0
  9 45 CONSTANT "-"  69 CONSTANT "E"
 10
 11 32 CONSTANT SPC
 12
 13 -->

SCR # 90
  0 ( FLT PKG - <-S LEFT SHIFT  )
  1 ( WORD2 WORD1 CY0 . . . 2*[WORD2 WORD1] CY1 )
  2 CODE <-S  H POP  L A MOV  RAR            ( GET CARRY TO A CY )
  3      H POP  W POP                        ( GET TWO WORDS )
  4      W' A MOV  RAL  A W' MOV             ( SHIFT LEFT THRU CY )
  5      W  A MOV  RAL  A W  MOV
  6      L  A MOV  RAL  A L  MOV
  7      H  A MOV  RAL  A H  MOV
  8      W  PUSH   H PUSH                    ( BACK TO STACK )
  9      0 A MVI  RAL  A L MOV  0 H MVI      ( SAVE CY AND .. )
 10      H PUSH  NEXT JMP                   ( PUT IT ON STK   )
 11
 12
 13 -->

SCR # 91
  0 ( FLT PKG - S-> RIGHT SHIFT, DBL-SHFTR, DBL-SHFTL, 10X )
  1 ( WORD2 WORD1 CY0 . . . [WORD2 WORD1]/2 CY1 )
  2 CODE S->  H POP  L A MOV  RAR
  3   H POP  W POP
  4   H  A MOV  RAR  A  H MOV
  5   L  A MOV  RAR  A  L MOV
  6   W  A MOV  RAR  A  W MOV
  7   W' A MOV  RAR  A W' MOV
  8   W PUSH  H PUSH
  9   0 A MVI  RAL  A L MOV  0 H MVI
 10  H PUSH  NEXT JMP
 11 : DBL-SHFTL >R >R 0 <-S R> R> ROT <-S DROP ;
 12 : DBL-SHFTR 0 S-> SWAP >R SWAP >R S->
 13             DROP R> R> ;
 14 : 10X 2DUP >R >R 0 <-S <-S DROP R> R> D+ 0 <-S DROP ;
 15 -->

SCR # 92
  0 ( FLT PKG - DNORM NORMALIZES A DOUBLE WORD )
  1 0 VARIABLE EXP  0 VARIABLE SGN
  2 : DNORM EXP ! 0 FLAG !
  3    OVER OVER OR 0= IF 0 ELSE 1 ENDIF
  4    CASE 0 OF 0 1 FLAG !        ENDOF   ( DO NOTHING AT ALL )
  5         1 OF DUP 0 < IF 1 SGN ! DMINUS ELSE 0 SGN ! ENDIF
  6            BEGIN -1 EXP +! 0 <-S DROP DUP 0 <
  7                  IF 0 S-> DROP 1 FLAG ! 1 EXP +! ENDIF
  8       FLAG @ UNTIL SGN @ IF DMINUS ENDIF EXP @        ENDOF
  9    ENDCASE ;
 10 : D>F 31 DNORM ;  ( CNVTS A DBL WORD INTEGER TO FP FORMAT )
 11 1. D>F FCONSTANT F1   10. D>F FCONSTANT F10
 12 100. D>F FCONSTANT F100
 13
 14 -->

SCR # 93
  0 ( FLT PKG - VARIABLES & FDUP, F2DUP, FSWAP, FROT  )
  1 ( M@, M!, BUFF )
  2 4 BUFF WORD1
  3 : M! WORD1 2! ;   : M@ WORD1 2@ ;
  4 0 VARIABLE SIGN1  0 VARIABLE SIGN2
  5 0 VARIABLE EXP1   0 VARIABLE EXP2
  6 0 VARIABLE EXP-STR  0 VARIABLE EXP-SGN
  7 F0 FVARIABLE STK1   F0 FVARIABLE STK2
  8 F0 FVARIABLE STK3
  9 : FDUP STK1 F! STK1 F@ STK1 F@ ;
 10 : F2DUP STK1 F! STK2 F! STK2 F@ STK1 F@ STK2 F@ STK1 F@ ;
 11 : FROT STK1 F! STK2 F! STK3 F! STK2 F@ STK2 F@ STK3 F@ ;
 12 : FSWAP STK1 F! STK2 F! STK1 F@ STK2 F@ ;
 13
 14 -->

SCR # 94
  0 ( FLT PKG - FVAL & TO )
  1 0 VARIABLE %TO     : TO 1 %TO ! ;
  2 ( FVAL DEFINES A VARIABLE THAT RETURNS ITS VALUE RATHER )
  3 ( THAN ITS ADDRESS UNLESS IT IS PRECEEDED BY 'TO' -     )
  4 ( IN WHICH CASE IT STORES A VALUE TO THE VARIABLE       )
  5
  6 : FVAL <BUILDS HERE 6 ALLOT F!
  7        DOES> %TO @ IF F! 0 %TO ! ELSE F@ ENDIF ;
  8 -->
```

(Continued on page 26)

Floating-Point Program (fig-Forth)

(Listing continued, text begins on page 16)

```
SCR # 95
  0 ( FLT PKG - D* UNSIGNED DBL WORD MULTIPLY )
  1 ( USE ON POSITIVE DBL WORDS ONLY )
  2 0 VARIABLE LO1   0 VARIABLE LO2
  3 0 VARIABLE HI1   0 VARIABLE HI2
  4 0 VARIABLE R1    0 VARIABLE R2
  5 0 VARIABLE R3    0 VARIABLE R4
  6
  7 : D* HI2 ! LO2 ! HI1 ! LO1 !
  8       LO1 @ LO2 @ U* SWAP R1 ! 0
  9       HI1 @ LO2 @ U* D+
 10       HI2 @ LO1 @ U* D+ SWAP R2 ! 0
 11       HI1 @ HI2 @ U* D+ R4 ! R3 !
 12       R1 @ R2 @ R3 @ R4 @ ;
 13
 14 -->

SCR # 96
  0 ( FLT PKG - DDNORM QUAD NORMALIZE + #'S ONLY ! )
  1 ( WORD4 WORD3 WORD2 WORD1 EXP . . . )
  2 ( . . . WORD4 WORD3 WORD2 WORD1 EXP' )
  3 : DDNORM EXP ! 0 FLAG ! >R >R OVER OVER OR R> SWAP
  4           OVER OR R> SWAP OVER OR 0= IF 0 ELSE 1 ENDIF
  5      CASE 0 OF 0 1 FLAG ! ENDOF  ( DO NOTHING, ITS ZERO )
  6           1 OF BEGIN -1 EXP +! DBL-SHFTL DUP
  7                      0 < IF DBL-SHFTR 1 FLAG ! 1 EXP +! ENDIF
  8         FLAG @ UNTIL  EXP @
  9           ENDOF
 10     ENDCASE ;
 11
 12 -->

SCR # 97
  0 ( FLT PKG - F* )
  1
  2
  3
  4
  5
  6 : F* 0 DUP SIGN1  ! SIGN2 ! 1+ EXP !
  7       DUP 0 < IF 1 SIGN1 ! DMINUS ENDIF M! EXP +!
  8       DUP 0 < IF 1 SIGN2 ! DMINUS ENDIF M@
  9       D* EXP @ DDNORM EXP ! M! SWAP DROP
 10       0 < IF M@ 1. D+ DUP 0 < IF 0 S-> DROP 1 EXP +! ENDIF
 11           ELSE M@ ENDIF
 12       SIGN1 @ SIGN2 @ - ABS IF DMINUS ENDIF EXP @ ;
 13
 14 -->

SCR # 98
  0 ( FLT PKG - F/ )
  1 : DIV-0 CR ." DIVIDE BY ZERO " CR QUIT ;
  2 ( F2 F1 ... F2/F1 )
  3 : F/ 0 DUP SIGN1 ! SIGN2 ! MINUS EXP !
  4   OVER OVER OR 0= IF 0 ELSE 1 ENDIF
  5   CASE 0 OF 5 NDROP DIV-0                                    ENDOF
```

```
SCR # 102
  0 ( FINP CONTINUED )
  1 : FINP 0 FLAG ! 0 DUP EXP1 ! SGN-IN !
  2    0 0 BEGIN KEY DUP DUP EMIT
  3              CASE    "-"          OF 1 SGN-IN ! DROP 0 ENDOF
  4                   CNTRL-C         OF ABORT-FINP          ENDOF
  5                   SPC             OF DROP 1 FLAG !       ENDOF
  6                   "E"             OF GET-EXP             ENDOF
  7                 "0" "9"  RNG-OF 48 -                     ENDOF
  8                 NUM-ERR
  9              ENDCASE
 10   FLAG @ 0= IF >R 10X R> 0 D+ ENDIF
 11   FLAG @ UNTIL D>F SGN-IN @ IF >R DMINUS R> ENDIF
 12 -->
 13
 14
 15

SCR # 103
  0 ( FLT PKG - FINP  CONTINUED )
  1  EXP1 @ IF EXP-STR @ DUP
  2            CASE 0 <OF ABS 0 DO F.1 F* LOOP ENDOF
  3                 0 >OF 0 DO F10 F* LOOP     ENDOF
  4               DROP
  5            ENDCASE
  6         ENDIF ;
  7
  8 0 VARIABLE LOOP-STR
  9 : SHIFT DUP 0 < IF DMINUS LOOP-STR @ 0 DO 0 S-> DROP LOOP
 10                 DMINUS ELSE LOOP-STR @
 11                              0 DO 0 S-> DROP LOOP
 12                 ENDIF ;
 13
 14 : ADDER D+ 1 EXP +! 0 S-> IF 1. D+ ENDIF DUP 0 < IF 0 S-> DROP
 15         1 EXP1 +! ENDIF ;  -->

SCR # 104
  0 ( FLT PKG - F+ AND F- )
  1 : F+ 0 DUP SIGN1 ! SIGN2 ! ( INITIALIZE SIGN STORE )
  2    EXP1 ! DUP 0 < IF 1 SIGN1 ! ENDIF M! ( SAVE EXP & SIGN )
  3    EXP2 ! DUP 0 < IF 1 SIGN2 ! ENDIF
  4 ( NOW ALIGN THE MANTISSAS )
  5 EXP1 @ CASE EXP2 @  OF EXP1 @ EXP !          ENDOF
  6             EXP2 @ >OF EXP1 @ EXP2 @ - LOOP-STR ! SHIFT
  7                        EXP1 @ EXP ! ENDOF
  8             EXP2 @ <OF EXP2 @ EXP1 @ - LOOP-STR ! M@ SHIFT
  9                        M! EXP2 @ EXP ! ENDOF
 10        ENDCASE
 11 SIGN1 @ SIGN2 @ +
 12       CASE 2  OF DMINUS M@ DMINUS ADDER DMINUS ENDOF ( 2 - #'S )
 13            1  OF M@ D+  ENDOF ( + & - # )
 14            0  OF M@ ADDER  ENDOF  ( 2 + #'S )
 15       ENDCASE   -->

SCR # 105
  0 ( FLT PKG - F+ AND F- CONTINUED )
  1 EXP @  DNORM ;
  2
  3 : F- >R DMINUS R> F+ ;
  4
  5
  6 -->
```

```
  6         1 OF DUP 0 < IF 1 SIGN1 ! ELSE DMINUS ENDIF
  7           M! ( MAKE DIVISOR NEGATIVE AND SAVE IT )
  8           EXP +!  ( EXP2 - EXP1 TO EXP )
  9           DUP 0 < IF 1 SIGN2 ! DMINUS ENDIF ( MAKE DVDND + )
 10           >R >R 0 0 R> R>  ( 0 QUOTIENT TO STACK )
 11         ENDOF
 12   ENDCASE
 13
 14 -->

SCR # 99
  0 ( FLT PKG - F/ CONTINUED )
  1 31 FLAG !  ( MAX # OF SHIFT LEFTS )
  2 BEGIN
  3       M@ D+ DUP 0 < IF 0 ELSE 1 ENDIF
  4  CASE 0 OF >R >R 0 <-S DROP R> R> M@  DMINUS D+ 0 <-S DROP
  5           -1 FLAG +!
  6         ENDOF
  7       1 OF >R >R 1 <-S DROP R> R> 0 <-S DROP -1 FLAG +!
  8         ENDOF
  9  ENDCASE
 10 FLAG @ 0= UNTIL 2 NDROP EXP @ 1+ DNORM
 11 SIGN1 @ SIGN2 @ - ABS IF >R DMINUS R> ENDIF ;
 12
 13
 14
 15


SCR # 100
  0 ( FLT PKG - FINP )
  1 : ABORT-FINP DROP DROP DROP CR ." COMMAND ABORT " CR
  2              QUIT ;
  3 : ABORT-FINP2 DROP ABORT-FINP ;
  4
  5
  6 0 VARIABLE SGN-IN
  7
  8 F1 F10 F/ FCONSTANT F.1
  9
 10 -->



SCR # 101
  0 ( FLT PKG - FINP  CONTINUED )
  1 : NUM-ERR 4 NDROP  0 0 0 0 CR ." MANTISSA ERROR " CR
  2           ." REENTER MANTISSA " CR ;
  3 : EXP-ERR 3 NDROP  0 0 0 CR ." EXP ERROR " CR
  4           ." REENTER EXPONENT " CR ;
  5 : GET-EXP DROP 1 EXP1 ! 0 EXP-SGN !
  6     0 BEGIN KEY DUP DUP EMIT
  7             CASE   SPC OF DROP 1 FLAG ! ENDOF
  8                    "-"  OF 1 EXP-SGN ! DROP 0  ENDOF
  9                "0" "9" RNG-OF 48 -       ENDOF
 10           CNTRL-C  OF DROP ABORT-FINP2  ENDOF
 11         EXP-ERR ENDCASE
 12 FLAG @ 0= IF SWAP 10 * + ENDIF
 13 FLAG @ UNTIL
 14 EXP-SGN @ IF MINUS ENDIF EXP-STR ! ;
 15 -->
```

(Continued on top of page 26)

```
SCR # 106
  0 ( FLT PKG - ADJUST )
  1 0 VARIABLE FLG 0 VARIABLE EXPX
  2 : ADJUST  0 FLG ! 0 EXP-STR !
  3   BEGIN DUP
  4   CASE    0   OF F.1 F* 1 EXP-STR +!          ENDOF
  5           0  >OF F.1 F* 1 EXP-STR +!           ENDOF
  6           0  <OF DUP 5 + 0 > IF 1 FLG !
  7                 ELSE F10 F* -1 EXP-STR +! ENDIF ENDOF
  8   ENDCASE
  9   FLG @ UNTIL    EXPX !
 10   0 <-S DROP 0 FLG !
 11   BEGIN EXPX @
 12     CASE  0  OF 1 FLG !              ENDOF
 13           0 <OF 0 S-> DROP 1 EXPX +!       ENDOF
 14     ENDCASE
 15 FLG @ UNTIL 2. D+ 0 <-S DROP  ;  -->

SCR # 107
  0 ( FLT PKG - PRNT#, F., #OUT )
  1
  2 : #OUT 10. D* DROP 48 + EMIT -1 FLG +! ;
  3
  4 : PRNT# OVER OVER 10. D* DROP 0= IF -2 EXP-STR +! DROP DROP
  5         10. D* DROP DROP ELSE -1 EXP-STR +! DROP DROP ENDIF
  6          #OUT ." ."
  7   BEGIN #OUT FLG @ 0= UNTIL DROP DROP ;
  8
  9 : F. 1- EXP ! DUP 0 < IF DMINUS ." -" ENDIF
 10      OVER OVER OR
 11  0= IF 0 EXP-STR ! ." 0." 9 0 DO ." 0" LOOP DROP DROP
 12    ELSE EXP @ ADJUST 9 FLG ! PRNT# ENDIF
 13     1 SPACES ." EXP " EXP-STR @ . ;
 14
 15


SCR # 109
  0 ( FLT PKG - FINT  RETURNS INTEGER PORTION )
  1 ( FP# . . . INTEGER PORTION )
  2 : ROTATL >R >R 0 <-S R> SWAP R> SWAP <-S DROP ;
  3 : ROTATR 0 S-> SWAP >R SWAP >R S-> DROP R> R> ;
  4 : GET-INT  0 0 EXP @ 1+ 0 DO ROTATL LOOP
  5  BEGIN 0 <-S UNTIL 1 S->  S-> DROP
  6     >R >R DROP DROP
  7      R> R> SGN @ IF DMINUS ENDIF EXP @ ;
  8 : FINT 0 SGN ! DUP 1- 0 <
  9         IF DROP DROP F0
 10        ELSE EXP ! DUP 0 < IF 1 SGN ! DMINUS ENDIF
 11             GET-INT ENDIF ;
 12 ( FMOD RETURNS REMAINDER OF A DIVISION IN FP FORMAT )
 13 : FMOD F2DUP F/ FINT F* F- ;
 14
 15 -->

SCR # 110
  0 ( FLT PKG - CNSTS FOR SINE & COSINE )
  1 HEX ED53 6487 2 FCONSTANT PI
  2 AAAF AAAA FE FCONSTANT FA2
  3 434B 4444 FA FCONSTANT FA4
  4 FEA7 97F9 F4 FCONSTANT FA6
```

(Continued on next page)

Floating-Point Program (fig-Forth)

(Listing continued, text begins on page 16)

```
  5 A76C 5C5C EE FCONSTANT FA8
  6 AC0B 9959 E7 FCONSTANT FA10  DECIMAL
  7
  8
  9 1000. D>F FCONSTANT F1000
 10 360.  D>F FCONSTANT F360
 11 180.  D>F FCONSTANT F180
 12 90.   D>F FCONSTANT F90
 13 45.   D>F FCONSTANT F45
 14 -->

SCR # 111
  0 ( FLT PKG - SINE & COSINE, ARG IN DEGREES, 2*10↑-9 ERROR )
  1 F0 FVARIABLE FSINX
  2 F0 FVARIABLE FXSQR     F0 FVARIABLE FPWR
  3 : *XSQR FXSQR F@ FPWR F@ F* FDUP FPWR F! ;
  4 : TERM  F* FSINX F@ F+ FSINX F! ;
  5 : FSIN  F180 F/ PI F*          ( CNVT TO RADIANS )
  6         F1 FSINX F!            ( 1ST TERM )
  7         FDUP FDUP F* FDUP FXSQR F! FDUP FPWR F!
  8         FDUP FPWR F!           ( INITIALIZED )
  9         FA2 TERM *XSQR FA4 TERM *XSQR
 10        FA6 TERM *XSQR FA8 TERM *XSQR
 11        FA10 F* FSINX F@ F+ F* ;
 12
 13 -->

SCR # 112
  0 ( FLT PKG - SINE & COSINE CONTINUED )
  1 : SINE  F360 FMOD
  2         FDUP F<0 IF F360 F+ ENDIF
  3         FDUP F180 F> IF F180 F- FSIN FMINUS
  4                      ELSE FSIN ENDIF ;
  5
  6 : COSINE F360 FMOD F90 F+ SINE ;
  7
  8 -->



SCR # 113
  0 ( FLT PKG - FSQR  RETURNS SQUARE ROOT OF FP# )
  1 F0 FVARIABLE ORIG     F0 FVARIABLE ITER
  2 0 VARIABLE CNTSQR
  3
  4 : FSQR FDUP F<0 IF FDROP CR ." NEG SQR ROOT " CR ENDIF
  5        FDUP ORIG F! DUP 127 > IF 256 + 2 /
  6                               ELSE 2 / ENDIF
  7        ITER F! 5 CNTSQR !
  8        BEGIN ORIG F@ ITER F@ F/
  9              ITER F@ F+ 1- ITER F!
 10             CNTSQR @ 1- DUP CNTSQR !
 11 0 = UNTIL   ITER F@ ;
 12
 13
 14
 15 -->
```

```
  3
  4
  5
  6
  7
  8 0 VARIABLE FFLAG
  9 : SETF FDUP F<0 IF 1 FFLAG ! FMINUS
 10                 ELSE 0 FFLAG ! ENDIF ;
 11 : TESTF FFLAG @ IF FMINUS ENDIF ;
 12
 13
 14
 15 -->

SCR # 116
  0 ( FLT PKG - ARCSIN CONTINUED )
  1 F10M F10 F* FCONSTANT F100M
  2 F100M F10 F* FCONSTANT F1B
  3 F1B F10 F* FCONSTANT F10B
  4 1.57079631 D>F F100M F/ FCONSTANT A0
  5 -.214598802 D>F F1B F/ FCONSTANT A1
  6 .0889789874 D>F F10B F/ FCONSTANT A2
  7 -.0501743046 D>F F10B F/ FCONSTANT A3
  8 .0308918810 D>F F10B F/ FCONSTANT A4
  9 -.0170881256 D>F F10B F/ FCONSTANT A5
 10 .0066700901 D>F F10B F/ FCONSTANT A6
 11 -.0012624911 D>F F10B F/ FCONSTANT A7
 12
 13
 14
 15 -->

SCR # 117
  0 ( FLT PKG - ARCSIN CONTINUED )
  1 : STORE FSINX F@ F+ FSINX F! ;
  2 : SETUP FSWAP FDUP FROT F* FDUP ;
  3 : FARC1 FDUP F1 FSWAP F- FSQR
  4         FMINUS FACTOR F!
  5         A0 FSINX F!
  6         FDUP A1 F* STORE
  7         FDUP FDUP F* FDUP A2 F* STORE
  8         SETUP A3 F* STORE
  9         SETUP A4 F* STORE
 10        SETUP A5 F* STORE
 11        SETUP A6 F* STORE
 12        F* A7 F* STORE
 13        FSINX F@ FACTOR F@ F* PI/2 F+ ;
 14 : ARCSIN-RAD SETF FARC1 TESTF ;
 15 : ARCSIN-DEG ARCSIN-RAD 180/PI F* ;

144 LIST
SCR # 144
  0 ( DR. EAKER'S CASE CONSTRUCT WITH A SLIGHT MODIFICATION )
  1 : CASE ?COMP CSP @ !CSP 4 ; IMMEDIATE
  2 : (OF) OVER = IF DROP 1 ELSE 0 ENDIF ;
  3 : OF 4 ?PAIRS COMPILE (OF) COMPILE 0BRANCH
  4    HERE 0 , 5 ; IMMEDIATE
  5 : ENDOF 5 ?PAIRS COMPILE BRANCH HERE 0 , SWAP 2
  6         [COMPILE] ENDIF 4 ; IMMEDIATE
  7 : ENDCASE 4 ?PAIRS COMPILE DROP BEGIN SP@ CSP @ = 0 =
  8           WHILE 2 [COMPILE] ENDIF REPEAT CSP ! ; IMMEDIATE
  9
 10
```

```
SCR # 114
  0 ( FLT PKG - ARCSIN-RAD, ARCSIN-DEG 2X10EXP-8 ACCURACY )
  1 F1000 F10 F* FCONSTANT F10K
  2 F10K F10 F* FCONSTANT F100K
  3 F100K F10 F* FCONSTANT F1M
  4 F1M F10 F* FCONSTANT F10M
  5 F1000 FCONSTANT F1K
  6 F0 FVARIABLE FACTOR
  7 2. D>F FCONSTANT F2
  8 PI F2 F/ FCONSTANT PI/2
  9 F180 PI F/ FCONSTANT 180/PI
 10 PI F180 F/ FCONSTANT PI/180
 11 -->

SCR # 115
  0 ( FLT PKG - ARCSIN CONTINUED )
  1
  2
 11 -->

OK
145 LIST
SCR # 145
  0 ( THE <OF, >OF, AND RNG-OF EXTENSIONS )
  1 : (<OF) OVER > IF DROP 1 ELSE 0 ENDIF ;
  2 : <OF 4 ?PAIRS COMPILE (<OF) COMPILE 0BRANCH
  3     HERE 0 , 5 ; IMMEDIATE
  4 : (>OF) OVER < IF DROP 1 ELSE 0 ENDIF ;
  5 : >OF 4 ?PAIRS COMPILE (>OF) COMPILE 0BRANCH
  6     HERE 0 , 5 ; IMMEDIATE
  7 : RANGE >R OVER DUP R> 1+ < IF SWAP 1- > IF DROP 1 ELSE 0
  8         ENDIF ELSE DROP DROP 0 ENDIF ;
  9 : RNG-OF 4 ?PAIRS COMPILE RANGE COMPILE 0BRANCH HERE 0 , 5 ;
 10 IMMEDIATE
 11
 12 -->
OK
```

End Listing

Forth Programming Style

by N. Solntseff

N. Solntseff, Unit for Computer Science, McMaster University, Hamilton, Ontario L8S 4K1.

This note proposes a publication standard for Forth programs in order to increase the comprehensibility of Forth program listings in magazines. I have also found that the technique described below is of great help to me in maintaining and testing Forth programs.

Several articles and special issues of personal computer magazines have been devoted to Forth within the last eighteen months[1-3]. The version of Forth standardized by the Forth Interest Group (fig) is now available for all popular 8-bit microcomputers. As an indication of the growing visibility of Forth on the personal computer scene, I have counted at least four different versions of fig-Forth78 or fig-Forth79 being sold for Ohio Scientific, Inc. (OSI) computers. There is even a version being distributed in ROM chips.

The virtues of Forth can be summarized as follows:

(1) It provides a very high degree of man/machine interaction, so that the program development cycle is considerably shortened by comparison with conventional languages.

(2) The Forth System, as implemented on personal computers, is largely machine independent and extremely compact. For example, memory used by a typical OSI implementation is

Address Interpreter	–	180 bytes
Machine-code Forth primitives	–	1670 bytes
High-level Forth primitives	–	4590 bytes
Operating system interface	–	600 bytes
OS65D (OSI operating system)	–	3710 bytes
Disk buffers	–	2056 bytes
Total		12806 bytes

A 16 Kbyte system with disk can be used for application programming in Forth.

(3) The extensibility of Forth allows the application program to develop an application language which incorporates terms familiar to the end user. Hence, the latter will be able to read and understand his programs much more readily than those written in conventional languages.

The drawbacks of Forth arise from its power and versatility as well as, in part, from the fact that until recently very little attention has been paid to it in educational circles. The most serious drawbacks are:

(1) Forth allows the programmer unrestricted access to the underlying hardware, but offers almost no protection against misbehaving programs. Thus, the computer reset button is an indispensable tool in Forth program debugging, a feature it shares with assembly-language programming.

(2) Educational computer users generally have ignored Forth and, as a result, there are few textbooks which discuss programming in Forth and not merely its linguistic features.[4]

(3) Apart from CONSTANT and VARIABLE declarations, fig-Forth provides only tools for the construction of data structures. Every programmer has to invent his own set, so that cooperative programming is extremely difficult to manage.

(4) Forth routines communicate with one another largely by means of the parameter (or data) stack. This means that the programmer or the reader of a program has to keep a mental picture of the stack and its changes during execution in order to understand the behavior of a Forth routine (or word). Because of this, published algorithms in Forth tend to be rather cryptic in appearance, requiring a considerable mental effort to comprehend.

My solution to the last problem is very simple – let us reserve the left-hand portion of each line in a Forth screen to show the appearance of the data stack during the execution of the word being defined. Only the topmost few elements of the stack need to be shown, enough to provide a visual aid to the programmer and the reader. An added advantage of this is that it is no longer necessary to give very long comments in order to explain how a given word works.

An example of the proposed technique is shown in the accompanying figure which gives the definitions for two non-trivial data-type definitions. The first defines BUFFER to be a data type which, as soon as the buffer becomes empty, automatically requests the user to input the next line from the terminal. In use, a call to a variable of type BUFFER returns the next available character in the buffer or a carriage return if the buffer is empty. The second example defines a data type for the storage of symmetrical matrices where only half of the off-diagonal elements need actually be stored.

Some judgment must be exercised in breaking up a Forth definition into manageable lines, but this can be quickly learned if the following criterion is used: Can the line of Forth words be understood from an examination of the stack field and the accompanying comment field? If this is not possible, then the sequence of Forth words is too long and needs to be broken down further.

I have been using the above technique for the last five months and strongly recommend it to all Forth programmers. I would also like to see it adopted as a publication standard for Forth programs.

References

[1] *Byte*, Volume 5, No. 8 (August 1980), pp. 76-196.

[2] Gary Fierbach, "Forth – The Language of Machine Independence," *Computer Design*, Volume 20, No. 6 (June 1981), pp. 117-121.

[3] *Dr. Dobb's Journal*, Volume 6, No. 9 (September 1981), pp. 6-53.

[4] Brodie, Leo, *Starting Forth*, Prentice-Hall, 1981.

Editor's Reply:

The style Mr. Solntseff suggests is satisfactory, but we can imagine some useful variations. The right end of source lines can be used for stack comments instead of the left. Each word should have a comment giving the net stack inputs and outputs. DOES> words should have two such comments, one for the compile-time (after BUILDS>) and another for execute-time (after DOES>).

Generally, style is improved by (1) choosing good names for words, (2) using small, well-factored definitions, and (3) using only a few (fewer than five) parameters on the stack in a word. This reduces the need for comments.

DDJ

(Listing begins at right)

Style Examples
(fig-Forth)

```
SCR # 114
  0 ( Self-Filling Buffer --                              114/NS/14nov81)
  1 : :BUFFER
  2         <BUILDS       ( data structure: cba : fba : lba : buffer )
  3                 HERE 86 ERASE      ( clear space: 80 chars + ptrs)
  4                 HERE 86 + ,      ( current ptr <== last  byte + 1)
  5                 HERE  4 +     ( first-byte ptr <== first byte     )
  6                 HERE 82 + ,    ( last-byte ptr <== last  byte + 1)
  7                 80 ALLOT    ( allocate 80 byte + 6 bytes for ptrs)
  8           DOES>                  ( returns next char on data stack)
  9          ( adr) DUP @              ( get address of next character)
 10      ( adr cba) OVER 4 + @      ( check to see if buffer is empty)
 11 ( adr cba lba) = IF ." >>>"              ( if empty, issue prompt)
 12          ( adr)      DUP 6 + 80 EXPECT  ( get line from terminal)
 13          ( adr)      DUP 6 + OVER ! ( set cba to start of buffer)
 14          ( adr)   ENDIF
 15          ( adr) DUP @ C@ SWAP OVER -->   ( get current character)


SCR # 115
  0 ( Self-Filling Buffer -- continued                    115/NS/14nov81)
  1
  2   ( ch adr ch) 00 = IF    DUP 4 + @ SWAP !  ( if retn, reset cba)
  3           ( ch)           DROP 13      ( replace 00 by car. return
  4       ( ch adr)       ELSE 1 SWAP +!         ( else, increment cba)
  5           ( ch)       ENDIF ;        ( leave current char on stack)
  6
  7 ;S
  8
  9 (
 10           cba : address of current buffer position
 11           fba : address of first buffer position
 12           lba : address of last  buffer position + 1
 13 )
 14
 15


SCR # 116
  0 ( Data structure for symmetrical matrices             116/NS/14nov81)
  1
  2 : :TRIANGULAR       ( n is matrix size; i,j are row,col indices)
  3                    <BUILDS
  4                ( n)   DUP 1+ 2 */ 2 *      ( calc. # locs needed)
  5         ( 2n*n+1/2)   HERE OVER ERASE        ( clear memory area)
  6         ( 2n*n+1/2)   ALLOT                       ( and allocate)
  7                    DOES>
  8          ( i j adr)   ROT ROT OVER OVER
  9      ( adr i j i j)   < IF SWAP ENDIF 2 *  ( exchange i,j if i<j)
 10         ( adr i 2j)   SWAP DUP 1+ 2 */ 2 *   ( mi in a = a0+mi+j)
 11 ( adr 2j 2i*i+1/2)   +                    ( get addr of i-th row)
 12 ( adr 2i*i+1/2+2j)   +           ( get address of i,j-th element)
 13       ( array[i,j])  ;                  ( leave address on stack)
 14
 15 ;S

    McFORTH [11aug81] -- Unit for C.Sc., McMaster University
```

End Listing

A Relocating Loader
in Forth

If you now use Forth, you can appreciate loading a 10K byte screen editor (compiled size) as a binary image in 5 seconds, instead of compiling it from source text in 200 seconds. Not only does this loader save and load Forth words very quickly, it can load the binary image at a different machine address than it was originally compiled at. Pre-compiled modules of commonly-used words can be kept on the disk and linked onto the Forth dictionary in any order. If you're interested in the inner workings of Forth – especially the public domain Forth Interest Group (fig) variety – you'll find an enormous amount of information about the layout of Forth "dictionary" entries contained in this program.

This relocating loader was written in a fig-compatible version of the language (Z-80 Forth by Laboratory Microsystems). By far the major difference between fig-Forth and Z-80 Forth is in the way they treat disk files: fig-Forth maintains a disk format incompatible with CP/M, while Z-80 Forth uses CP/M files to store programs and data.

Operation of the Words GET and SAVE

To create a binary image for later recall, the desired Forth words are first compiled into the dictionary from their source text definitions. Once compiled, the binary image of that segment of the Forth dictionary is written to a file by typing:

SAVE <word><filename>

where <word> is the first Forth word of the module and <filename> is the name of the CP/M file that holds the image.

After the binary image has been created with SAVE, it can be recalled with the word GET. In operation, GET reads the binary image file and appends the Forth definitions contained in it onto the end of the current Forth dictionary. To load the definitions, type:

GET <filename>

where <filename> again represents the name of the CP/M file that contains the binary image.

by Joe Barnhart

Joe Barnhart, P.O. Box 4383, Santa Rosa, CA 95402.

As implemented on 8080 and Z-80 computers, Forth does not produce position-independent code. If a Forth program is compiled to run at one machine address and moved to another (or loaded at another), it will not run properly. After it loads a binary image, this loader modifies the Forth definitions so that they will operate properly at their new addresses. High-level ("colon") definitions, constants, variables, and <BUILDS DOES> definitions are all relocatable. Machine code definitions are not relocatable and this loader issues an error message if you try to save one in a binary image file.

SAVE

The simplest of the two definitions presented here, SAVE performs the following steps:

1. Check the dictionary segment for machine language definitions. Issue an error message if any are found.
2. Save the dictionary segment in a CP/M file.
3. Save four numbers that are used in the relocation process.

Checking the dictionary for machine language (or "code") definitions is straightforward. ?CODE-DEF checks a single word and reports an error if the word is a code definition. NO-CODE-DEFS calls the word ?CODE-DEF repeatedly until all of the Forth definitions in the dictionary segment have been examined.

When SAVEd, the dictionary segment is written into a CP/M file separate from other Forth text and data. In Z-80 Forth, a filename is parsed from the console input stream by the word FILENAME. OPEN/MAKE tries to open a file of a given name. If the file doesn't exist, OPEN/MAKE creates it. Finally, WRITESEC writes one sector (128 bytes) to the file sequentially.

Before the file is closed, four numbers are written into it that allow the dictionary segment to be relocated when it is read again. The numbers are:

1. The size of the dictionary segment in bytes. It is used to allot enough space for the definitions when they are loaded again.

2. The byte offset to the Link Field Address (LFA) of the last word in the dictionary segment. It allows the loader to find the last definition in the segment.

3. The present compiled address of the first word in the dictionary segment. With it, the loader calculates the difference between the original compiled address and the loaded address.

4. The byte offset to the last vocabulary defined in the dictionary. When loading the segment again, it indicates whether or not any vocabularies are in the segment.

GET

Loading the segment into the dictionary again is a little more complicated than saving it. First, the CP/M file containing the segment is read into memory and appended to the dictionary. The four numbers are then recovered from the end of the file and the segment is relocated to run at its new address.

Opening and reading the CP/M file is done in much the same way as the file was written. FILENAME again parses the name of the file from the console input stream. OPEN and READFILE do what their names imply. After the file has been read, the word GETADDR recovers the four numbers from the end of the CP/M file. It stores the difference between the segment's original compiled address and its loaded address in the variable OFFSET. The original compiled address of the first definition in the segment is stored in the variable FIRST.NFA. The size of the segment, the point to the last vocabulary, and the address of the last word in the segment are left on the stack.

Central to the operation of the loader is the definition RE-LINK. All of the actual relocation is performed by invoking RE-LINK with the address of the last word in the segment on the stack. RE-LINK has two jobs: (1) to re-establish the linked list of words that make up the Forth dictionary, and (2) to add any offsets necessary to the definitions themselves to make them operate at a new address.

Restoring the linked list of words in the new portion of the dictionary is easy. The offset between the original compiled address and the loaded address of the words is the same for all of the words in the dictionary segment. By adding this offset to the linked list pointer in each Link Field Address, the integrity of the linked list is restored.

RE-LINK examines each definition and "patches" it according to the type of Forth word it is. Colon definitions, <BUILDS DOES> definitions, and vo-

cabularies require special treatment. Colon definitions are treated by the word COL.DEF and <BUILDS DOES> definitions are treated by DOES.DEF. Vocabularies, which are a special kind of <BUILDS DOES> definition, are treated by the word VOC.DEF.

RE-LINK is a recursive Forth definition. By using the word SMUDGE at the beginning of the definition and again at the end, the normal Forth mechanism for preventing "accidental" recursion is bypassed. Forth can have "tree structured" vocabularies within its dictionary (Figure 1). To link and relocate all definitions, each vocabulary must be followed to its end. When it detects a new vocabulary, RE-LINK treats it (with a call to VOC.DEF) and calls itself to relocate the newly found vocabulary. When the new vocabulary has been completely relocated, RE-LINK continues where it left off (Figure 2).

COL.DEF relocates a single colon definition to run at a different address than its compiled address. High-level Forth definitions are a list of the addresses that are executed when the word is invoked. In the simplest case, a high-level definition that is composed entirely of words defined in the main Forth dictionary (not the loaded segment) needs no relocation. On the other hand, if some of the definitions are defined in the loaded dictionary segment, the addresses must be offset by the difference between the loaded and compiled address. Two Forth words deserve special treatment: LIT and (."). LIT pushes a 16-bit literal value onto the stack when executed. The literal value, which is compiled into the definition after LIT, is skipped by the loader to prevent it from accidentally being "relocated" and its value changed. Similarly, (.") is followed by a string which, like the value after LIT, must be skipped by the loader.

DOES.DEF relocates the DOES> portion of a <BUILDS DOES> definition. First, it checks the address of the DOES> part to see if it is defined in the dictionary segment just loaded. If it is, the address of the DOES> word is offset by the difference between the compiled and loaded addresses.

The pointers contained in the vocabulary definition are modified by the word VOC.DEF. A vocabulary definition contains a pointer to the last word defined in the vocabulary, and a pointer to the previous vocabulary defined in the dictionary. Each of these are checked and offset if they are within the loaded segment. The word VOC.DEF leaves the address of the last word defined in the vocabulary on the stack, for the recursive RE-LINK call.

After the dictionary segment has been RE-LINKed, all that remains is to update the Forth variable VOC-LINK and allot space in the dictionary for the definitions.

Suggested Modifications

There are two modifications to perform if this program is to be run on an 8080 fig-Forth system. For one, GET and SAVE must be modified to use a standard Forth screen file instead of a separate CP/M file. Reserve the first line of each screen as a comment line to avoid sending machine code to the console whenever the screen file is INDEXed.

A more subtle difference between fig-Forth and Z-80 Forth is found in the words BRANCH and 0BRANCH. Both of these words cause the Forth interpreter to jump to an address, which is compiled just after the BRANCH or 0BRANCH instruction. In standard 8080 fig-Forth, the address is coded as a relative address. With Z-80 Forth, however, the address is an absolute address. Instead of relocating the address, modify the loader to skip over it.

(Listing begins on page 38)

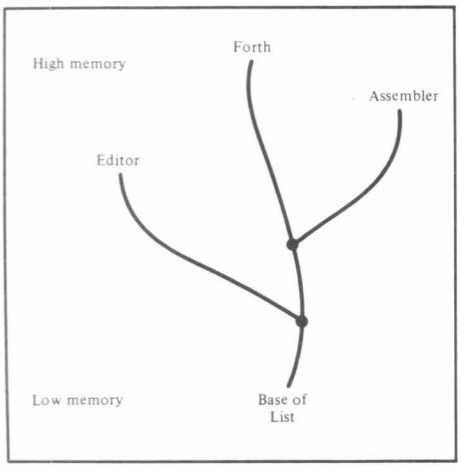


Figure 1.
Forth can have "tree structured" vocabulary.

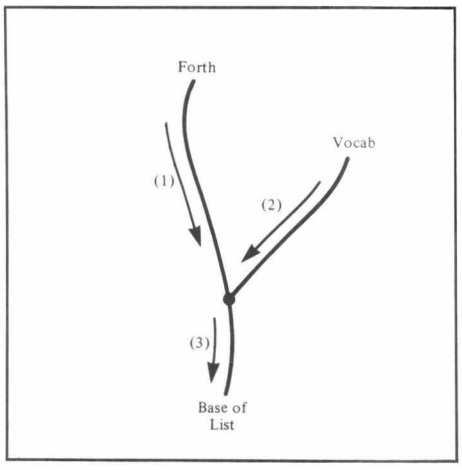


Figure 2.
After relocating a new vocabulary, RE-LINK continues where it left off.

Relocating Loader (Z-80 Forth)

(Text begins on page 34)

```
   Screen # 6
 0 ( Error messages for Z-80 Assembler )
 1 16 bit register not allowed
 2 8 bit register not allowed
 3 index register not allowed
 4 immediate data value not allowed
 5 missing source register
 6 missing destination register
 7 illegal operation
 8 illegal operand
 9 instruction not implemented
10 illegal destination register
11 illegal source register
12 illegal condition code
13 register mismatch
14
15
```

```
   Screen # 7
 0 ( GET and SAVE for Z-80 FORTH      Joe Barnhart, 3/03/82 )
 1  FORTH DEFINITIONS DECIMAL         : TASK ;
 2  128 CONSTANT B/SEC
 3  : FCB       ( FCB name .... ) ( ---  fcbaddr )
 4    <BUILDS 37 HERE OVER ERASE ALLOT DOES> ;
 5
 6  : SET-DMA      ( addr  --- )
 7    26 SWAP FDOS DROP DROP ;
 8
 9  : READSEC      ( fcbaddr addr  ---  errorcode )
10    SET-DMA 20 SWAP FDOS DROP ;
11
12  : WRITESEC     ( fcbaddr addr  ---  errorcode )
13    SET-DMA 21 SWAP FDOS DROP
14    IF 8 ERROR ENDIF ;
15 -->
```

```
   Screen # 8
 0 ( GET and SAVE:  OPEN/MAKE, CLOSE, ?SAMEVOC )
 1 : OPEN/MAKE      ( fcb-address ---   )
 2   DUP 15 SWAP FDOS DROP 255 =     ( try to open file first )
 3   IF 22 SWAP FDOS DROP 255 =      ( if no file then create one )
 4     8 ?ERROR                      ( no directory space, error )
 5   ELSE DROP ENDIF ;               ( normal return )
 6
 7 : CLOSE          ( fcb-address ---  )
 8   16 SWAP FDOS DROP 255 =         ( try to close file )
 9   8 ?ERROR ;
10
11 : ?SAMEVOC       (  ---  )
```

```
 9      2DUP WRITESEC               ( write a sector )
10      B/SEC + DUP                 ( increment the DMA address )
11      HERE >                      ( repeat until addr>DP )
12   UNTIL
13 -->
14
15
```

```
   Screen # 12
 0 ( GET and SAVE:  SAVE, cont. )
 1
 2   DUP B/SEC ERASE                 ( nice and clean... )
 3   HERE R - OVER ! DUP             ( store the dict. ptr. )
 4   2+ LATEST PFA LFA R - OVER !    ( the LFA of the last word )
 5   2+ R OVER !                     ( the NFA of the first word)
 6   VOC-LINK @ R> - SWAP 2+ !       ( and the VOC-LINK )
 7   2DUP WRITESEC                   ( write it to the file )
 8   DROP CLOSE                      ( and close it )
 9 ; -->
10                    SAVE <word> <filename>
11
12  This word, SAVE, saves a binary image of the FORTH words from
13  <word> to the end of the dictionary.  Use it, along with GET,
14  to save and load FORTH vocabularies at high speed. See screen
15  157 for more detail.
```

```
   Screen # 13
 0 ( GET and SAVE:  OPEN, READFILE )
 1   52 LOAD          ( case construct by Charles Eaker )
 2 : OPEN             ( fcb-addr  ---  )  ( followed by filename )
 3   DUP FILENAME                        ( get filename that follows )
 4   15 SWAP FDOS DROP 255 =             ( try to open file )
 5   8 ?ERROR ;                          ( abort if doesn't exist )
 6
 7 : READFILE       ( fcb-addr  dma-addr  ---  new-dma-addr )
 8   BEGIN
 9     2DUP READSEC                      ( read one sector )
10     0=                                ( if <>0, then end of file )
11   WHILE                               ( while NOT end of file )
12     B/SEC +                           ( increment dma address )
13   REPEAT                              ( repeat until file end )
14   SWAP DROP B/SEC - ;                 ( drop fcb-addr )
15 -->
```

```
   Screen # 14
 0 ( GET and SAVE:  constants, DOES.DEF )
 1          0          VARIABLE OFFSET       ( between read & comp )
 2          0          VARIABLE FIRST.NFA    ( first NFA in module )
 3  ' VLIST CFA     @ CONSTANT   COL.CFA     ( CFA of 'docol' )
 4  ' FORTH CFA DUP @ CONSTANT  DOES.CFA     ( CFA of 'dodoes' )
 5          2+      @ CONSTANT   VOC.PFA     ( PFA of vocabulary def)
 6  ' ;S    CFA       CONSTANT  SEMI.CFA     ( CFA of 'semis' )
 7  ' (.")  CFA       CONSTANT  DOTQ.CFA     ( CFA of 'dot quote' )
 8  ' LIT   CFA       CONSTANT   LIT.CFA     ( CFA of 'lit' )
```

```
12   CONTEXT @ CURRENT @ = 0=       ( are the context and current )
13   24 ?ERROR ;                    ( vocabularies the same? )
14 -->
15
```

```
   Screen # 9
 0 ( GET and SAVE:  FILENAME )
 1 ( This word courtesy of Ray Duncan, Laboratory Microsystems )
 2 : FILENAME      ( fcb-address FILENAME <nametext> )
 3   DUP 37 ERASE 1+ DUP 11 BLANKS ( clear file control block )
 4   BLK @ IF BLK @ BLOCK ELSE TIB @ ENDIF IN @ + ( text addr )
 5   8 0 DO  ( first format filename ) DUP C@ DUP ( next char )
 6   46 = IF DROP LEAVE ELSE  ( quit if dot )
 7   DUP 33 < IF DROP LEAVE   ( or if space or null )
 8   ELSE ROT DUP I + ROT SWAP C! SWAP 1+ 1 IN +! ENDIF ENDIF LOOP
 9   DUP C@ 46 = ( is there an extension? )
10   IF SWAP 8 + SWAP 1+ 1 IN +! 3 0 DO  ( yes format it )
11   DUP C@ ( fetch next char. )
12   DUP 33 < IF DROP LEAVE  ( quit if null or space )
13   ELSE ROT DUP I + ROT SWAP C! SWAP 1+ 1 IN +! ENDIF LOOP
14   ENDIF ( done with extension )
15   DROP DROP  ( clean up stack ) ;   -->
```

```
   Screen # 10
 0 ( GET and SAVE:  BINFILE, ?CODE-DEF, NO-CODE-DEFS )
 1  FCB BINFILE
 2  : ?CODE-DEF             ( PFA  ---  T/F )
 3    CFA DUP @                      ( get the CFA of target word )
 4    - MINUS 2 = ;                  ( CFA is addr+2? <true if code>)
 5
 6  : NO-CODE-DEFS          ( PFA  ---  )
 7    >R LATEST PFA 0                ( start at last word, push zero)
 8    BEGIN                          ( check all words down to PFA )
 9      OVER ?CODE-DEF OR            ( is it a code def? )
10      SWAP LFA @ PFA SWAP          ( get next word's PFA )
11      OVER R <                     ( and see if we're done )
12    UNTIL
13    SWAP DROP R> DROP              ( and if we are... clean stack )
14    5 ?ERROR ;                     ( signal error if code defs )
15 -->
```

```
   Screen # 11
 0 ( GET and SAVE:  SAVE )
 1 : SAVE      ( SAVE  word  filename.ext )
 2   ?SAMEVOC                       ( same context and current? )
 3   [COMPILE] ' DUP NFA >R         ( get the name of the word, PFA)
 4   DUP  NO-CODE-DEFS              ( check for any code defs )
 5   BINFILE FILENAME               ( get filename, allocate FCB )
 6   BINFILE OPEN/MAKE              ( open or make a file )
 7   NFA BINFILE SWAP               ( prepare to write a sector )
 8   BEGIN
```

(Continued on top of page 38)

```
 9    HEX   A081      CONSTANT   END.VOC   ( vocab. intersection )
10                    DECIMAL
11 : DOES.DEF       ( LFA  ---  LFA )
12   DUP 2+ 2+ DUP @                ( get PFA of does part )
13   FIRST.NFA @ >                  ( is it in relocated section? )
14   IF OFFSET @ SWAP +!            ( if so, offset it...)
15   ELSE DROP ENDIF ; -->          ( ...else drop the address )
```

```
   Screen # 15
 0 ( GET and SAVE:  COL.DEF )
 1 : COL.DEF DUP    ( LFA  ---  LFA )
 2   BEGIN                          ( add offsets to colon words )
 3     2+ DUP @ SEMI.CFA -          ( while not end of word )
 4   WHILE
 5     DUP @ CASE                   ( offset the word if needed )
 6       DOTQ.CFA OF 2+ DUP C@ + 1- ENDOF ( skip text string )
 7       LIT.CFA  OF 2+ ENDOF       ( skip literal constant)
 8       FIRST.NFA @ >              ( is def in new module? )
 9       IF OFFSET @ OVER +! ENDIF  ( if so, offset it )
10     DUP ENDCASE                  ( endcase consumes one item )
11   REPEAT                         ( until 'semis' )
12   DROP ;                         ( drop address of 'semis' )
13 -->
14
15
```

```
   Screen # 16
 0 ( GET and SAVE:  VOC.DEF, GETADDR )
 1 : VOC.DEF        ( LFA  ---  LFA  voc-LFA )
 2   DUP 8 + OFFSET @ OVER +!       ( offset the last NFA in vocab )
 3   DUP 2+ @ FIRST.NFA @ >         ( does voc-link --> module? )
 4   IF OFFSET @ OVER 2+ +! ENDIF   ( yes, offset it )
 5   @ PFA LFA ;                    ( fetch voc-LFA for nxt re-link)
 6
 7 : GETADDR        ( addr  ---  size  voc-link  last-LFA )
 8   DUP @ SWAP                     ( size of module in bytes )
 9   2+ DUP @ HERE + SWAP           ( new LFA of last word )
10   2+ DUP @ DUP FIRST.NFA !       ( first NFA before relocation )
11   HERE SWAP - OFFSET !           ( calculate offset at new addr )
12   2+ @ SWAP ;                    ( last vocabulary in module )
13 -->
14
15
```

```
   Screen # 17
 0 ( GET and SAVE:  RE-LINK )
 1 : RE-LINK [ SMUDGE ] BEGIN       ( first-LFA  ---  ) ( recursive)
 2     OFFSET @ OVER +!             ( patch the LFA )
 3     DUP 2+ @ CASE                ( get CFA of definition )
```

(Continued on next page)

Relocating Loader
(Z-80 Forth)

(Listing continued, text begins on page 34)

```
 4       COL.CFA  OF COL.DEF ENDOF ( colon definition? )
 5       DOES.CFA OF              ( DOES> definition? )
 6         DUP 2+ 2+ @ VOC.PFA =  ( is it a vocabulary? )
 7           IF VOC.DEF RE-LINK   ( yes, re-link it recursive )
 8           ELSE DOES.DEF ENDIF  ( else regular DOES> def )
 9         ENDOF
10     ENDCASE
11     DUP @ DUP HERE <           ( until LFA < HERE )
12     SWAP @ END.VOC = OR 0=     ( or end of vocabulary )
13   WHILE
14     @ PFA LFA                  ( follow linked list to next )
15   REPEAT -->                   ( continued next screen...)
```

```
  Screen # 18
 0 ( GET and SAVE:  RE-LINK, cont. )
 1   DUP @ HERE <  IF LATEST      ( last def in module, so...)
 2   SWAP ! ELSE DROP ENDIF ;     ( ...link it to the LATEST def )
 3   SMUDGE                       ( un-smudge header )
 4 -->
 5                          GET <filename>
 6
 7   This word retrieves any binary image file saved with the SAVE
 8   command.  After the binary image is found and loaded from the
 9   disk, the FORTH words contained in it are "relocated" to run
10   at their new address.  All forth high-level defs, variables,
11   constants, DOES> defs, and vocabularies are accomodated.  Only
12   code defs are un-relocatable.  Laboratory Microsystem's Z-80
13   assembler, for instance, takes about 105 seconds to LOAD from
14   a disk but only 4 seconds to GET.
15
```

```
  Screen # 19
 0 ( GET and SAVE:  GET )
 1 : GET           ( GET <filename> )
 2   ?SAMEVOC                     ( same context and current? )
 3   BINFILE OPEN                 ( open the filename )
 4   BINFILE HERE READFILE        ( read entire file at DP )
 5   GETADDR                      ( get the addresses from file )
 6   DUP RE-LINK                  ( relocate all definitions )
 7   2+ 2+ NFA CONTEXT @ !        ( update the addr in LATEST )
 8   DUP 0 >                      ( is the last vocab in module? )
 9     IF HERE + VOC-LINK !       ( if so, update VOC-LINK )
10     ELSE DROP ENDIF            ( otherwise, drop it )
11   ALLOT                        ( extend the DP for new defs )
12   BINFILE CLOSE                ( close the file )
13 ; ;S
14
15
```

```
  Screen # 20
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
10
11
12
13
14
15
```

```
  Screen # 52
 0 ( Case statement, by Charles E. Eaker )
 1 ( from FORTH DIMENSIONS II/3 page 37  )
 2 : CASE ?COMP CSP @ !CSP 4 ; IMMEDIATE
 3 : OF 4 ?PAIRS COMPILE OVER COMPILE = COMPILE 0BRANCH
 4   HERE 0 , COMPILE DROP 5 ; IMMEDIATE
 5 : ENDOF 5 ?PAIRS COMPILE BRANCH HERE 0 ,
 6   SWAP 2 [COMPILE] ENDIF 4 ; IMMEDIATE
 7 : ENDCASE 4 ?PAIRS COMPILE DROP BEGIN SP@
 8   CSP @ = 0= WHILE 2 [COMPILE] ENDIF
 9   REPEAT CSP ! ; IMMEDIATE
10
11
12
13
14
15
```

```
  Screen # 4
 0 ( System messages )
 1 empty stack
 2 dictionary full
 3 has incorrect address mode
 4 isn't unique
 5 no code definitions allowed
 6 disc range ?
 7 full stack
 8 disc error !
 9
10
11
12
13 BASE must be DECIMAL
14 missing decimal point
15 Z-80 FORTH
```

End Listing

For the H-19

Forth Screen Editor

This article describes an editor written in Forth Interest Group Forth, and used for editing Forth source programs. It uses the built-in editing capabilities of a Heath H-19 terminal. It was designed for use by students just learning Forth, and has several features that I consider particularly important for the novice. These are:

1. The use of function keys and information displayed permanently on the terminal to make reference to external documentation unnecessary;

2. Use of terminal hardware for character and line insertion and deletion, which gives immediate feedback to the user; and

3. Inclusion of features designed to encourage and simplify program documentation.

I use Forth in several environments. At home, where I have a memory-mapped video display on my homebrew CP/M system, one of my first projects after getting Forth up from the fig listings was the creation of a screen-oriented editor to replace the simple fig editor. One of the joys of Forth programming became the ability to shove my program text around, knowing that what I saw on the screen was exactly what would appear in the final listing.

One of the other environments in which I use Forth is at Hampshire College, where I am a professor. I soon got Forth going on the CP/M equipment that was available to me there. My nice editor wasn't easily moved to that system, however. The Heath H-19 terminal used at the College precluded direct manipulation of text in memory. The Heath terminal has a lot of built-in editing features of its own, but for lack of time to work on the problem I continued to use the fig editor. My inability to use the H-19's character and line insertion and deletion features was a constant irritant, however.

For an intensive January term course exploring the techniques used to implement high-level interpreters and compilers, I wanted to get a number of students working with Forth. Although Forth is not sophisticated as high-level languages go, it is a complete system, and concepts of operating systems, interpreters, and compilers are all part of Forth. A second reason for choosing Forth for this purpose was that all the inner workings of the Forth system are visible – Forth is like a plastic mechanical toy in which you can see the gears turning inside.

by A.S. Woodhull

Albert S. Woodhull, School of Natural Science, Hampshire College, Amherst, MA 01002.

The fig editor, like CP/M's ED, uses various letters of the alphabet as commands. My experience in using ED, and my observations of student's experiences with it, convinced me that better editing was needed. With neither of these editors do you see the effects of your command string until after you have pressed Return. A skilled user of such a system can get good results, but for the novice it means working with an open users manual and frustrations that divert attention from the real task.

The editor presented in Listing 1 is my response to the situation. It was put together in a fairly short period of time, but it incorporates what I learned from almost a year of use of my memory-mapped editor. I consider it a particularly good editor for student use, because all the documentation needed to make it work is actually on the terminal keys or on the H-19 screen at all times, and the effect of all actions are instantly visible.

The Editor is invoked by typing

n ED

where n is the number of the first screen to edit. This clears the screen and sets up a border around the text area. Current documentation strings which identify the project, the programmer, and the date are then displayed and the user is asked if they are to be kept or changed. Following the response to this query the first screen to be edited is fetched.

Figure 1 shows the appearance of the screen during an editing session. Since Forth works with screens of 16 lines of 64 characters and the Heath terminal has 24 lines of 80 characters plus a 25th line for status, there is room for presentation of additional information. The editor runs with the numeric keypad in its alternate mode, enabling the built-in editing functions. The four arrow keys for cursor control are self-explanatory, but the insertion and deletion labels seemed a bit cryptic and are further explained on the screen. The 25th line on the screen furnishes labels for the user-definable function keys. From left to right these functions are:

I.D. – inserts the current project name, the user's initials, and the date, on line zero. This line, often called the index line, is conventionally devoted to comments by Forth programmers. The relabelling does not affect the middle part of the line, so the characters in positions 20 through 47 are available to identify the individual screens.

Border – the border around the text area gets messed up by deletions and insertions, and can be redrawn by this function.

< Erase – this erases from the cursor to the left edge.

IC- – this terminates the character insertion mode.

Erase > – this erases the right-hand end of a line.

Quit – this quits without replacing the screen on the disk.

SCR# – this assigns a new screen number, where the edited screen will be stored on the disk after an End command. This allows copying and moving single screens.

End – this terminates the edit and places the edited screen on the disk.

Editing is itself a fairly simple process – text is typed at the position where it is wanted, using the cursor control keys and the insertion/deletion keys. In this mode, all characters except escape sequences are immediately echoed to the screen. The program only has to filter out the escape sequences that correspond to commands. The sequence generated by the HOME key is also trapped and direct cursor addressing is used to put the cursor on its new "home" within the text area.

Following interception of the code corresponding to the End key, the H-19 is commanded to transmit the entire screen. Since the H-19 is not smart enough to transmit only part of the screen, this required some Forth words to ignore the superfluous parts of the screen. This is the only part of the editor where I found Forth's ability to compile assembly language CODE routines necessary.

After either an End or Quit command, the functions of the left- and right-arrow keys and the HOME key are redefined. The operator is prompted to use the arrows to select either the next higher or lower numbered screen for editing, or to use the HOME key to enable

entry of a new screen number where editing should recommence. Hitting any other command key (i.e., one generating an escape sequence) at this point terminates the editing session.

The listing of the program should be easy to follow if you are used to Forth programs, which are (or should be) designed from the top down but listed from the bottom up. The last screen in the listing defines ED, the outer loop which does the setup and queries for the next screen to edit after each Quit or End. The next-to-last screen defines EDEXEC, which gets the screen to be edited from the disk and checks to see if keyboard input is an escape sequence to be processed further or a character to be echoed. Most of the Forth words defined in the editor are named with the Heath mnemonics for the H-19 escape sequences. The few CODE routines that are used are written as hex values to be entered into the Forth dictionary with comma words (, and C,), so a Forth assembler is not needed.

There are a few items in the listing that are not part of the standard fig-Forth 8080 version 1.1. I added a word ?DECIMAL to check whether I was in decimal mode before attempting a disk access, since the CP/M disk routines used by Forth do not respond gracefully when asked to fetch the 256th screen on a 250K disk. I also have added the string routines described by Ralph Deane in *DDJ* #50. I extended these by adding a word #IN which uses the SINP (string input) word to fetch a string from the keyboard which is then converted to a number. Those whose Forths differ from mine will find it easy to define similar words.

I hope readers will find the ideas in this editor useful. Although it does not do everything one might wish – in particular it cannot move blocks of text smaller than a screen – it does put the considerable power of the H-19 terminal to good use. I am also convinced that a less-powerful editor that is easy to use may be more desirable for some purpose, particularly in teaching, than a powerful editor the novice can use only with the help of a reference manual. Many of us who have become skilled in working with computers tend to forget the feelings of frustration created by the arbitrary nature of the abbreviated commands to which our tools respond.

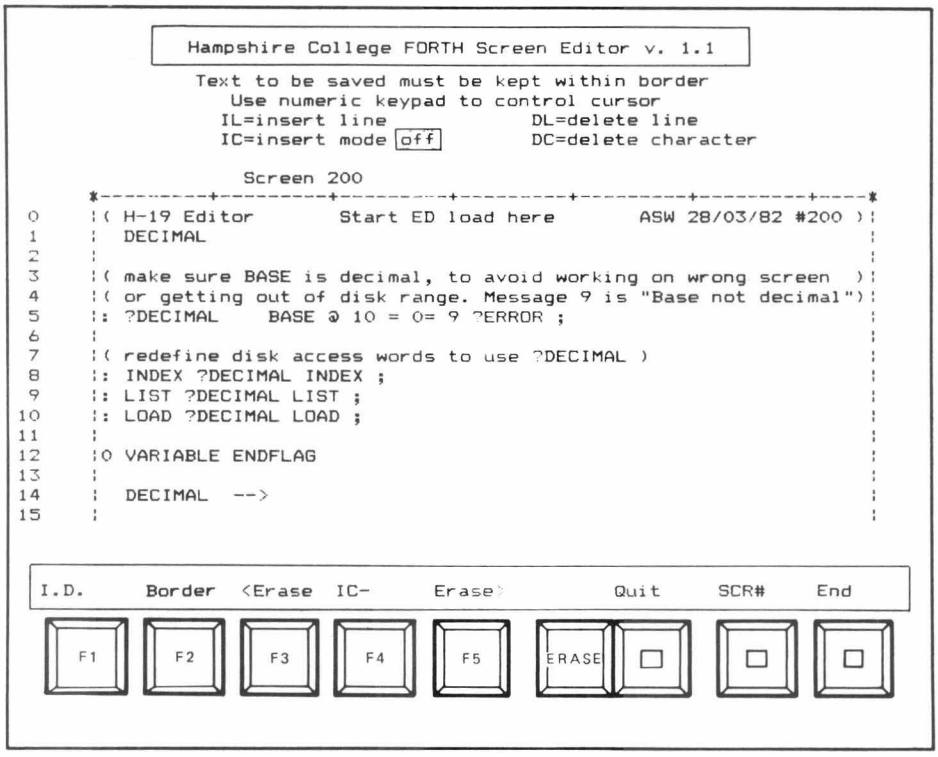


Figure 1.

Appearance of screen during editing. Box indicates reversed video.

I am willing to copy the Forth source code for this editor onto an IBM format 8" single-density floppy disk for anyone who will send a blank disk with a suitable return mail package and adequate return postage. ■■J

(Listing begins on page 45)

H-19 Editor (fig-Forth)

(Text begins on page 42)

```
SCR * 200
  0 ( H-19 EDITOR        START ED LOAD HERE        ASW 28/03/82 *200 )
  1   DECIMAL
  2
  3 ( MAKE SURE BASE IS DECIMAL, TO AVOID WORKING ON WRONG SCREEN  )
  4 ( OR GETTING OUT OF DISK RANGE. MESSAGE 9 IS "BASE NOT DECIMAL")
  5 : ?DECIMAL    BASE a 10 = 0= 9 ?ERROR ;
  6
  7 ( REDEFINE DISK ACCESS WORDS TO USE ?DECIMAL )
  8 : INDEX ?DECIMAL INDEX ;
  9 : LIST ?DECIMAL LIST ;
 10 : LOAD ?DECIMAL LOAD ;
 11
 12 0 VARIABLE ENDFLAG
 13
 14   DECIMAL  -->
 15


SCR * 201
  0 ( H-19 EDITOR        HEATH ESCAPE SEQUENCES    ASW 28/03/82 *201 )
  1  HEX   ( THE HXXX WORDS ARE HEATH COMPANY MNEMONICS )
  2 : HXMP 1B EMIT 23 EMIT ;        ( TRANSMIT PAGE )
  3 : HDCA 1B EMIT 59 EMIT 20 + EMIT 20 + EMIT ;
  4                                 ( X Y ___      SET CURSOR POS. )
  5 : HEKS 1B EMIT 74 EMIT ;        ( SHIFTED KEYPAD MODE )
  6 : HXKS 1B EMIT 75 EMIT ;        ( EXIT KEYPAD SHIFTED )
  7 : HEBL 1B EMIT 6F EMIT ;        ( ERASE START OF LINE )
  8 : HEOL 1B EMIT 4B EMIT ;        (    "     END     "   )
  9 : HSM  1B EMIT 78 EMIT 30 + EMIT ; ( SET MODE         N ___ )
 10 : HCD  1B EMIT 45 EMIT ;        ( CLEAR DISPLAY )
 11 : HERV 1B EMIT 70 EMIT ;        ( REVERSE VIDEO ON )
 12 : HXRV 1B EMIT 71 EMIT ;        (    "       "  OFF )
 13 : HEIM 1B EMIT 40 EMIT ;        ( ENTER INSERT MODE )
 14 : HERM 1B EMIT 4F EMIT ;        ( EXIT INSERT MODE )
 15 : HSCP 1B EMIT 6A EMIT ;        ( SAVE CURSOR POSITION )  -->


SCR * 202
  0 ( H-19 EDITOR        SPECIAL ESCS              ASW 28/03/82 *202 )
  1 HEX
  2 : HRCP 1B EMIT 6B EMIT ;        ( RESTORE CURSOR )
  3 : HDK  1B EMIT 7D EMIT ;        ( DISABLE KEYBOARD )
  4 : HEK 1B EMIT 7B EMIT ;         ( ENABLE KEYBOARD )
  5
  6 0 VARIABLE ICFLAG     ( REMEMBER IF INSERT MODE ON/OFF )
  7
  8 ( PUT UP A NOTE IN REVERSE VIDEO TO TELL USER IF CHARACTER )
  9 ( INSERTION IS CURRENTLY ON OR OFF )
 10 : IC- HERM HSCP 21 4 HDCA HERV ." OFF " HXRV HRCP 0 ICFLAG ! ;
 11 : IC+ HSCP 21 4 HDCA HERV ." ON  " HXRV HRCP HEIM 1 ICFLAG ! ;
 12
 13 : HOME 8 8 HDCA ;  ( HOME FOR FORTH IS ON THE 16X64 WINDOW )
 14
 15     DECIMAL   -->

SCR * 203
  0 ( H-19 EDITOR        DOCUMENTATION TOOLS       ASW 28/03/82 *203 )
  1   DECIMAL
```

(Continued on top of page 45)

```
  2 ( STRING ROUTINES FROM DDJ *50 LEAVE ADDRESS OF FIRST BYTE AND )
  3 ( CURRENT LENGTH OF STRING ON STACK WHEN STRING NAME INVOKED )
  4 ( STRING DECLARATIONS SET MAXIMUM LENGTH, CREATE NULL STRINGS )
  5 9 STRING DATE$       3 STRING NAME$      20 STRING PROJ$
  6 32 STRING TEMP$
  7
  8 : SCRID  ( PUT SCREEN NUMBER ONTO SCREEN )
  9        66 8 HDCA " *" TYPE DECIMAL SCR a 3 .R "  )" TYPE ;
 10
 11 : SCMSG  0 6 HDCA IC- HEOL 20 6 HDCA ." SCREEN " SCP a . ;
 12
 13 : DOID  IC- 8 8 HDCA  ." ( " PROJ$ TYPE
 14         53 8 HDCA NAME$ TYPE    57 8 HDCA DATE$ TYPE
                                                15          SCRID HOME

SCR * 204
  0 ( H-19 EDITOR        DOCUMENTATION PROMPTS     ASW 28/03/82 *204 )
  1  DECIMAL
  2
  3 : DOCINFO   ( PROMPT FOR AND GET STRINGS FOR INDEX LINES )
  4     0 6 HDCA HEOL ." CURRENT I.D.: " DOID  16 6 HDCA
  5     ." PRESS  UP ARROW TO KEEP, ANY OTHER COMMAND KEY TO CHANGE"
  6     KEY DROP KEY 65 = 0=  ( DROP ESC, TEST FOR "A" )
  7   IF
  8      0 6 HDCA HEOL ." ENTER CURRENT DATE AS  DD/MM/YY "
  9              TEMP$ SINP 12 TEMP$ LEFT$ DATE$ S!
 10      0 6 HDCA HEOL ." ENTER YOUR INITIALS AS XYZ "
 11              TEMP$ SINP 3 TEMP$ LEFT$ NAME$ S!
 12      0 6 HDCA HEOL ." ENTER TITLE, UP TO 20 CHARACTERS: "
 13              TEMP$ SINP 20 TEMP$ LEFT$ PROJ$ S!
 14   ENDIF 0 6 HDCA HEOL ;
 15             DECIMAL -->


SCR * 205
  0 ( H-19 EDITOR        SCREEN NUMBERING          ASW 28/03/82 *205 )
  1  DECIMAL
  2
  3
  4 : NRSCRN    (  ___            PROMPT, GET NUMBER, STORE IN SCR )
  5         DECIMAL  IC-
  6         0 6 HDCA HEOL ." ENTER SCREEN NUMBER: " HXKS *IN HEKS
  7         SCR !  SCRID SCMSG HOME ;
  8
  9                                    DECIMAL -->
 10
 11          THIS ALLOWS FOR COPYING AND MOVING SCREENS ON DISK.
 12 EACH BLOCK IS RE-IDENTIFIED BEFORE BEING REPLACED, USING THE
 13 CURRENT VALUE OF SCR.  THE SCR* KEY CAN THUS BE USED TO STORE
 14 AN EDITED SCREEN AT A DIFFERENT LOCATION FROM WHERE IT WAS FIRST
 15 READ.

SCR * 206
  0 ( H-19 EDITOR        SCREEN HANDLING DOC.      ASW 28/03/82 *206 )
  1 -->
  2 THE STANDARD F.I.G. FORTH DISK INTERFACE DEALS WITH 128 BYTE
  3 SINGLE DENSITY DISK SECTORS, 8 OF WHICH MAKE UP AN EDITING
  4 SCREEN.  FORTH'S DISK BUFFER AREA CONSISTS OF 8 BLOCK BUFFERS,
  5 EACH OF WHICH HOLDS ONE DISK SECTOR.  THIS ALLOWS FORTH TO
  6 DO MANY ACTIONS WHICH DO NOT REQUIRE FETCHING A WHOLE SCREEN,
  7 SUCH AS PRINTING MESSAGES STORED ON DISK, VERY EFFICIENTLY, BUT
  8 IT IS AWKWARD FOR USE WITH A SCREEN EDITOR, SINCE THE INDIVIDUAL
  9 SECTOR BUFFERS ARE SEPARATED BY 4 BYTES WHICH HOLD THE BLOCK
 10 NUMBERS AND UPDATE STATUS.
 11    TO BEGIN EDITING A SCREEN SCET IS USED TO GET THE 8 SECTORS
 12 INTO THE FORTH BLOCK BUFFERS. THE BUFFERS ARE FIRST CLEARED AND
 13 INITIALIZED SO THE 8 SECTORS WILL BE ENTERED IN ORDER.
```

(Continued on next page)

H-19 Editor
(fig-Forth)

(Listing continued, text begins on page 42)

```
 14    VGET PUTS THE TEXT INTO THE EDITING WINDOW ON THE H-19
 15 TERMINAL SCREEN.


SCR # 207
  0 ( H-19 EDITOR        SCREEN HANDLING DOC.      ASW 28/03/82 #207 )
  1 -->
  2                                    THE H-19 CAN NOT BE COM-
  3 MANDED TO TRANSMIT ONLY THE SCREEN WINDOW, SO VPUT USES THE CODE
  4 ROUTINES DROPKEY AND EXPCD TO IGNORE SOME PARTS OF THE TRANS-
  5 MISSION AND TO PLACE THE OTHER PARTS IN THE BLOCK BUFFERS. AN
  6 ARRAY BFLINES IS USED TO HOLD THE ADDRESSES AT WHICH EACH LINE
  7 IS TO BE STORED.   SPUT RENUMBERS ALL THE BUFFERS AND MARKS THEM
  8 ALL AS UPDATED, THEN  CALLS FLUSH TO REPLACE THEM TO THE DISK.
  9
 10         WHEN THE TEXT IS SENT TO THE H-19 SCREEN IT IS POSITION-
 11 SO A SINGLE TAB FROM THE LEFT MARGIN OF THE SCREEN WILL PUT THE
 12 CURSOR ON THE LEFT MARGIN OF THE TEXT.
 13
 14
 15



SCR # 208
  0 ( H-19 EDITOR        SCREEN HANDLERS           ASW 28/03/82 #208 )
  1
  2   HEX
  3
  4 : SBLCKS     ( ___ LAST+1 1ST    DISK BLOCK FOR CURRENT SCR )
  5              SCR @ 8 * DUP 8 + SWAP ;
  6
  7 : SGET       ( SCREEN# ___       GET FROM DISK TO BUFFERS )
  8     SCR ! FIRST DUP USE ! [ 84 7 * ] LITERAL + PREV !
  9     EMPTY-BUFFERS          SBLCKS DO I BLOCK DROP LOOP ;
 10
 11    DECIMAL -->
 12
 13   THESE WORDS ARE FOR F.I.G. FORTH 8080 DISTRIBUTION VERSION,
 14 SET FOR 8 SINGLE DENSITY SECTORS OF 128 BYTES EACH PER EDITING
 15 SCREEN.


SCR # 209
  0 ( H-19 EDITOR        SCREEN HANDLERS           ASW 28/03/82 #209 )
  1
  2  HEX
  3 : BUFADDR       ( N ___ ADDR       GET ADDR OF BUFFER START )
  4                 ( NOTE: ADDRESS RETURNED IS LOC OF BLOCK #, DATA
  5                 ( BEGINS AT ADDR+2 )
  6     B/BUF 4 + * FIRST + ;
  7
  8 : UPDATE-ALL    ( MARK ALL BUFFERS TO BE SAVED )
  9     B/SCR 0  DO    I BUFADDR DUP @ 8000 OR SWAP !    LOOP ;
 10
 11 : IDBUFS        ( FIX BLOCK #S IN BUFFERS, IN CASE A MOVE WANTED
 12     B/SCR 0  DO  SCR @ B/SCR * OFFSET @ + I + I BUFADDR ! LOOP ;
 13
 14 : SPUT          ( PUT BACK ON DISK )
 15     IDBUFS UPDATE-ALL FLUSH ;                    DCIMAL -->


SCR # 210
  0 ( H-19 EDITOR        SCREEN HANDLERS           ASW 28/03/82 #210 )
 11          40 EXPCD  8 DROPKEYS  ( GET LINE, DROP  8  AFTER TEXT )
 12        LOOP
 13        IDBUFS SPUT          ( FIX IDS, UPDATE AND FLUSH TO DISK )
 14        HEX ;                              ( R-ENABLE KEYBOARD )
 15                                          DECIMAL -->


SCR # 214
  0 ( H-19 EDITOR        SETUP 1                   ASW 28/03/82 #214 )
  1   HEX
  2 : BOX         ( MAKE A BORDER AROUND THE TEXT AREA )
  3               7 7 HDCA              ( SET AT UPPER LEFT CORNER )
  4               ." *----------+---------+---------+---------+"
  5               ." ----------+---------+----*              "
  6               ( PUT LINE NUMBERS OUTSIDE LEFT BORDER )
  7               18 8 DO 0 I HDCA I 8 - 3 .R 7 I HDCA 7C EMIT  LOOP
  8               ( CLEAR SCREEN OUTSIDE RIGHT BORDER )
  9               18 8 DO 48 I HDCA HEOL 7C EMIT LOOP ;
 10
 11   DECIMAL
 12  -->
 13
 14
 15


SCR # 215
  0 ( H-19 EDITOR        SETUP 2                   ASW 28/03/82 #215 )
  1   HEX
  2
  3 : TITLE HERV
  4     10 0 HDCA ." HAMPSHIRE COLLEGE FORTH SCREEN EDITOR V. 1.1 "
  5          HXRV
  6     10 1 HDCA ." TEXT TO BE SAVED MUST BE KEPT WITHIN BORDER"
  7     13 2 HDCA ." USE NUMERIC KEYPAD TO CONTROL CURSOR "
  8     12 3 HDCA ." IL=INSERT LINE             DL=DELETE LINE"
  9     12 4 HDCA ." IC=INSERT MODE             DC=DELETE CHARACTER" ;
 10
 11 : LINE25      1 HSM HERV 0 18 HDCA HEOL  5 18 HDCA
 12 ."  I.D.  BORDER <ERASE  IC-  ERASE>       QUIT    SCR#    END "
 13           HXRV   ;
 14                                                DECIMAL  -->
 15


SCR # 216
  0 ( H-19 EDITOR        SETUP 3                   ASW 28/03/82 #216 )
  1   HEX
  2 : FORMAT      ( SHOW PROMPT INFO, SHIFT KEYPAD, GET SCREEN )
  3       BOX  TITLE IC- LINE25 HEKS ;
  4
  5 : SETUP  HCD FORMAT ;
  6
  7 ( DECODE WHETHER TO ADVANCE, GO BACK, OR BREAK ORDER )
  8 : NEWSCR  ( C ___      C IS 44 FOR <--, 43 FOR -->, 48 FOR HOME )
  9       DUP 48 = IF DROP NRSCRN
 10          ELSE 44 = IF -1 ELSE 1 ENDIF  SCR @ + SCR ! ENDIF ;
 11     DECIMAL -->
 12
 13
 14
 15



SCR # 217
  0 ( H-19 EDITOR        USER-DEFINED KEYS         ASW 28/03/82 #217 )
  1    HEX
  2 : 2NULLS  DROP DROP 0 0 ;
  3 : EMSG  SCMSG ." WILL BE UPDATED " ;
  4 : QMSG  SCMSG ." UNCHANGED " ;
```

```
 1
 2  HEX
 3
 4 : VGET   ( PRINT SCREEN BUFFER IN WINDOW AREA ON H-19 )
 5          HDK          ( DISABLE KEYBOARD )
 6          8 0 DO       ( 2 LINES ARE IN EACH OF THE 8 BUFFERS )
 7            I BUFADDR 2 + DUP 40 8 I 2 * 8 + HDCA TYPE
 8            40 + 40 8 I 2 * 9 + HDCA TYPE
 9          LOOP
10          8 8 HDCA     ( HOME CURSOR )
11          HEK ;        ( ENABLE KEYBOARD )
12  DECIMAL -->
13
14
15


SCR # 211
 0 ( H-19 EDITOR       SCREEN HANDLERS          ASW 28/03/82 #211 )
 1   HEX
 2 ( THESE CODE ROUTINES ARE USED TO RECEIVE THE SCREEN )
 3                  ( TRANSMITTED BY THE H-19 TERMINAL )
 4 CODE DROPKEYS  ( N ___          IGNORE N CHARS )
 5     D1 C, D5 C, CD C, 164E , D1 C, 1B C, 7B C, B2 C, HERE 8 -
 6      C2 C, , C3 C, 145 , SMUDGE
 7
 8 CODE EXPCD   ( ADDRESS N ___       GET N BYTES, STOW AT ADDR )
 9           D1 C, E1 C,   ( COUNT IN DE, ADDR IN HL )
10           E5 C, D5 C, CD C, 164E , D1 C, E1 C,  ( GET CHAR )
11           77 C, 23 C, 1B C,   ( STOW CHAR, INC ADDR, DEC COUNT )
12           7B C, B2 C,   ( TEST  COUNT )  HERE 0C - C2 C, ,
13           C3 C, 145 ,   ( JMP NEXT )  SMUDGE
14
15     DECIMAL   -->


SCR # 212
 0 ( H-19 EDITOR       SCREEN HANDLERS          ASW 28/03/82 #212 )
 1  HEX
 2
 3 ( THE FOLLOWING CREATES AN ARRAY OF 16 ELEMENTS, HOLDING THE )
 4 ( ADDRESSES TO WHICH THE 16 LINES OF THE EDITED SCREEN MUST BE )
 5 ( RETURNED )
 6
 7 FIRST 2 + DUP VARIABLE BFLINES 40 + DUP , 44 + DUP , 40 + DUP ,
 8      44 + DUP , 40 + DUP , 44 + DUP , 40 + DUP ,
 9      44 + DUP , 40 + DUP , 44 + DUP , 40 + DUP ,
10      44 + DUP , 40 + DUP , 44 + DUP , 40 + DUP ,
11
12 -->
13
14
15


SCR # 213
 0 ( H-19 EDITOR       SCREEN HANDLERS          ASW 28/03/82 #213 )
 1  HEX
 2
 3 : VPUT              ( RECEIVE A PAGE BACK TO THE BLOCK BUFFERS )
 4             ( SKIP 8 LINES-- EXTRA CHARS FOR REV. VIDEO ON/OFF )
 5       HDK              ( DISABLE KEYBOARD DURING TRANSMISSION )
 6       HXMP               ( COMMAND H-19 TO SEND WHOLE SCREEN )
 7       [ 8 50 * 8 + ] LITERAL DROPKEYS  ( IGNORE TOP OF SCREEN )
 8       10 0 DO
 9         8 DROPKEYS                ( DROP 8 CHARS BEFORE TEXT )
10         BFLINES I I + + @         ( GET ADDRESS FOR NEXT LINE )
```

(Continued on top of page 46)

```
 5 : HF1 DOID 2NULLS ;                    ( REPLACE INDEX LINE )
 6 : HF2 IC- BOX 2NULLS ;                 ( REDRAW THE BORDER )
 7 : HF3 HEBL 2NULLS ;                    ( INSERT MODE OFF )
 8 : HF4 IC- 2NULLS ;                     ( ERASE LEFT )
 9 : HF5 HEOL 2NULLS ;                    ( ERASE RIGHT )
10 : HF8 NRSCRN 2NULLS ;                  ( RENUMBER SCREEN )
11 : HF7 2NULLS 1 ENDFLAG ! QMSG ;        ( QUIT W/O SAVING )
12 : HF9  2NULLS 1 ENDFLAG ! EMSG VPUT ;  ( END, SAVE UPDATED VERS )
13  DECIMAL -->
14
15


SCR # 218
 0 ( H-19 EDITOR       DECODE ESCAPE SEQ.       ASW 28/03/82 #218 )
 1 ( S=1,I.D. T=2,FORMAT U=3,IC- P=BLUE,QUIT P=WHITE,END )
 2    HEX
 3
 4 : ESCKEY
 5    KEY  ( GET CHAR AFTER ESC )
 6     ( EACH OF THE FOLLOWING MUST LEAVE 2 CHARS )
 7    DUP 40 = IF IC+ 2NULLS ENDIF   DUP 48 = IF HOME 2NULLS ENDIF
 8    DUP 53 = IF HF1 ENDIF     DUP 54 = IF HF2 ENDIF
 9    DUP 55 = IF HF3 ENDIF     DUP 56 = IF HF4 ENDIF
10    DUP 57 = IF HF5 ENDIF
11    DUP 50 = IF HF7 ENDIF     DUP 51 = IF HF8 ENDIF
12    DUP 52 = IF HF9 ENDIF
13    SWAP EMIT EMIT ;
14
15    DECIMAL -->


SCR # 219
 0 ( H-19 EDITOR       EDITOR INNER LOOP        ASW 28/03/82 #219 )
 1    HEX
 2 : EDEXEC
 3      0 6 HDCA HEOL
 4      SCR @ SGET VGET  SCMSG HOME    0 ENDFLAG !
 5      BEGIN   ( ECHO ALL EXCEPT ESCAPES TO TERMINAL )
 6        KEY DUP  1B =     ( IS IT ESC ? )
 7          IF
 8            ICFLAG @ IF IC- ENDIF ( ALWAYS TURN OFF INSERT MODE )
 9            ESCKEY   ( QUIT OR END WILL RETURN TRUE FLAG )
10          ELSE
11            EMIT
12          ENDIF
13        ENDFLAG @
14      UNTIL ;
15  DECIMAL  -->


SCR # 220
 0 ( H-19 EDITOR       ED OUTER LOOP            ASW 28/03/82 #220 )
 1    HEX
 2 : ED        ( SCREEN# ___ )
 3      ?DECIMAL SCR !
 4      SETUP DOCINFO
 5      BEGIN
 6         EDEXEC
 7         0 6 HDCA HEOL ." NEXT: -->     PREVIOUS: <--     NEW #:"
 8                    ." HOME    OR ANY OTHER COMMAND KEY TO QUIT"
 9         BEGIN    KEY 1B =   UNTIL
10         KEY DUP DUP DUP 43 = SWAP 44 = OR SWAP 48 = OR
11         IF   NEWSCR 0  ELSE   DROP 1  ENDIF
12      UNTIL HCD HXKS ;
13  DECIMAL   ;S
14
15
```

End Listing

Z8000 Forth

by Lou Odette

Lou Odette, College of Engineering, Boston University, 110 Cummington St., Boston, MA 02215.

This version of Forth was written as a tool with which to develop a Data Base Management System (DBMS). The DBMS forms the core of a special purpose communications system built around the Z8000 microprocessor. Two features of Forth make it particularly attractive for this level of systems programming. The first feature is Forth's dual nature as an interpreter and compiler. This makes program development and testing interactive and consequently faster than the usual development cycle of edit, assemble, link and test. The second feature is the ability to extend the Forth instruction set by adding operators, functions and data structures as needed. Extensibility means that the language can be tailored to the application. The DBMS mentioned above is designed as a virtual machine whose instruction set is implemented as Forth words. Forth is used in this context as a high-level assembly language.

The Forth described here includes many of the words in the Forth-79 standard. The major omission is the word set dealing with mass storage access. Installation of this Forth on a particular system using the Z8000 requires customization of the terminal I/O subroutines. However, there are only two subroutines involved, one that gets an input line from the console, and another that sends an output line to the console. The details of register usage by these routines are given in the program listing.

R. G. Loeliger motivated this Z8000 Forth project with an outstanding presentation of the features of Forth-like languages in his book *Threaded Interpretive Languages*. My design for Z8000 Forth is a descendant of the Z80 threaded interpretive language (TIL) discussed by Loeliger. In order to take advantage of addressing and stack manipulation features of the Z8000, I have written a different address interpreter and modified Loeliger's dictionary format. In particular, since the Z8000 can use any of its general purpose registers as a stack pointer, it is convenient to treat the instruction register as a stack pointer and the Forth word definition as a stack. The address interpreter just pops compilation addresses and data as needed. In addition the indirect addressing mode provided by the Z8000 allows one less level of indirection than used by Loeliger's Z80 TIL. Thus, the compilation address of a word definition can be the address of executable machine code. [*This is called "direct threaded code."*] In the Z80 TIL, the compilation address contains a pointer to the machine code. [*This is called "indirect threaded code."*]

This article assumes familiarity with the Forth language and with some of the details of the implementation of the language on another machine. I expect that if readers have installed fig-Forth on their own machines they shouldn't have too much difficulty following the source code presented here. Readers of Loeliger's book should also be able to follow the source readily, and can use the Z8000 code as an additional example of a threaded language. Details of the Z8000 implementation of Forth are discussed below.

Memory Allocation The language resides in the lower memory address space and occupies less than 3K bytes. System variables occupy memory beneath the system code. Data on the system data area can be accessed with Forth words like CONTEXT, which gets the address of the context vocabulary from the system data area and leaves it on the data stack. Since data is indexed by position relative to the start of the system data area, the order of the system data must be maintained.

When Forth is started, the initialization routine initializes the data and return stacks to the upper end of the address space, and the dictionary pointer is set to the first free byte after the language. The data and return stack build down in memory while the language builds upward.

Dictionary Format Dictionary entries for each word in the language consist of a header and a code body located in contiguous memory locations. I use a header that is six bytes long for each dictionary entry: the first four bytes consist of the number of characters in the word name (one byte), followed by the ASCII code for the first three characters of the word. Thus, two words are the same if they have both the same number of characters in their names, and the same first three characters. The final two bytes of the header is the link field and contains the address of the header of the preceding dictionary entry. To construct headers and clearly identify dictionary entries in the program listing, I use the macro construct HEADER.

The code body of each Forth word is located after its header. Since the address of the code body of a dictionary entry is the compilation address of the word, the compilation address is equivalent to the word definition.

The first four bytes of the code body always contain executable machine code, while the remainder of the code body is either machine code or a list of the compilation addresses of other Forth words, intermixed with data. In a word definition, the first address available for the storage of compilation address and data (compilation address + 4) is called the parameter field address. In this implementation, compilation address and data in a word definition occupy two bytes of memory.

The headers of the words in the dictionary form a linked list that is searched to locate the compilation address of a Forth word. The last dictionary entry is indicated by a zero link address, and a zero link signals the dictionary search algorithm that the search has terminated unsuccessfully. If the search routine successfully locates a Forth word in the dictionary, it places the address of the code body (compilation address) on the data stack.

Address Interpreter The address interpreter interprets the sequence of compilation addresses in a word definition by executing the definition at each one. Since each compilation address in a definition may point to another list of compilation addresses, a Forth word definition has a tree structure. It is the job of the address interpreter to traverse the tree to the leaves, which are always machine code routines. In other words, only native machine code can be executed, and the address interpreter is used to control the order of execution of machine code routines.

As the address interpreter traverses the tree defined by the list of compilation addresses in a word definition, it maintains an instruction register that points to the next element in the current list (the compilation addresss of the next word to be executed) while execution begins at the compilation address of the current word. Execution begins when the

Z8000 program counter is set to the compilation address of the word.

If the code body of a word contains machine code, execution proceeds. However, if the code body is a list of compilation addresses and data, the code at the compilation address of the word calls a routine in the address interpreter called COLON. The COLON routine pushes the contents of the instruction register on the return stack and then loads the instruction register with the parameter field address of the word. The address interpreter then executes the word whose compilation address is at the parameter field address. The effect is that COLON nests down to begin execution of a lower level routine.

Threaded lists end with the address of the address interpreter routine SEMI. When SEMI is executed, the top of the return stack is popped to the instruction register, de-nesting one level.

The address interpreter routine that executes the next compilation address is called NEXT. NEXT consists of two instructions: (1) a pop instruction which uses the instruction register as the stack pointer to both increment the instruction register and move the compilation address to a Z8000 general register, and (2) an indirect jump to the compilation address of the word to be executed. Since machine code routines are the leaves of the tree, they terminate with the instructions of the NEXT routine.

Text Interpreter The text interpreter is a Forth word that serves as the system executive. My text interpreter follows Loeliger's design: The interpreter brings in a line at a time from the terminal and parses the line a word at a time. Each word in the input line is a sequence of characters terminated by one or more spaces. The word is moved to the dictionary in extended header format: the length of the character string (one byte) followed by the complete ASCII code for the string.

Once the word is moved to the end of the dictionary, a search routine in the text interpreter searches the linked list of dictionary entry headers for the first header that matches the input word and, if a match is found, leaves the compilation address of the matching word on the data stack. The search routine always leaves a flag on the stack, indicating success or failure of the search.

Following a successful search, the next word executed in the text interpreter depends on the system mode. If the system is in execute mode, the word is executed. If the system is in compile mode, and the word is not an immediate word, the compilation address on the data stack is added to the list of compilation addresses and data at the end of the dictionary, and the dictionary pointer is incremented. Immediate words are compiler directives and are executed directly in compile mode.

I use a separate vocabulary to implement immediate words. This vocabulary is searched by the search routine if the word is not found in the context vocabulary and the system is in compile mode.

If the search of the headers is unsuccessful, the text interpreter tries to convert the input word at the end of the dictionary into a number, using the current number base. If the number conversion is successful and the system is in execute mode, the number is placed on the data stack. When in compile mode, the number, together with a literal handler, is placed in the list of compilation addresses and data being compiled.

If a word cannot be found in the dictionary and cannot be converted to a number, the character string is passed to the terminal output routine, followed by a ":". The text interpreter ceases to parse the present input line, and instead jumps back to execute the word that accepts a new line from the terminal.

Defining Words Defining words are used to make new words from previously defined words. The character ":" (colon) is a defining word that, when executed, creates a dictionary header for the word following ":" in the input stream, and then sets the compile mode. The compilation address following the new header contains the instruction CALL COLON. The remainder of the input character stream is then compiled by the text interpreter as described above.

New word definitions initiated by a colon are terminated by the immediate word ";" (semicolon). Since ";" is an immediate word it is executed in the compile mode. Execution of ";" encloses the compilation address of the address interpreter routine SEMI in the dictionary and sets the system to execute mode. With the exception of ":", defining words return the system to execute mode on completion.

Examples of other defining words are the words "VARIABLE" and "CONSTANT". Like ":" these words create dictionary entries for the next word in the input stream. The compilation address of "CONSTANT" contains a call to a routine that pushes the contents of the parameter field address onto the stack. The compilation address of words created by "VARIABLE" contain a call to the address interpreter routine NEXT. This leaves the parameter field address on the data stack and causes the address interpreter to proceed with execution of the next word.

The word pair CREATE . . . DOES> is used in the definition of new defining words. When "CREATE" is executed, a header for the next word in the input character stream is added to the dictionary. Words following CREATE in the defining word definition can be used to add data and compilation addresses of other words to the dictionary. Execution of DOES> places a call to a special machine language routine in the compilation address of the new word. This causes the machine code to be executed when the new word is interpreted by the address interpreter. The machine language routine places the parameter field address of the word on the data stack and sets the instruction register to the address of the word following DOES> in the defining word definition.

Like any other word, a defining word containing CREATE . . . DOES> is a list of compilation addresses and data. However, this list is divided into defining-time code (between CREATE and DOES>) and run-time code (words following DOES>). Defining-time code is executed when the defining word is executed. Run-time code is executed when the word built by the defining word is executed.

Control Structures With the exception of EXECUTE, the control words are immediate words that are executed during compilation. The effect of execution is to add both the compilation address of system directives and relative branch data to the list of addresses and data being constructed by the compiler. For example, during compilation of a word containing the word pair BEGIN . . . UNTIL, execution of BEGIN places the address of the next free dictionary byte on the data stack. Execution of UNTIL adds the compilation address of the system directive *sysend* to the dictionary, pops the data stack and subtracts the stack entry from the address of the next free dictionary

byte. The result (two bytes) is added to the dictionary as a relative branch constant.

When *sysend* is executed during the execution of the word containing BEGIN . . . UNTIL, the top of the data stack is popped and compared to zero. If equal to zero, the relative branch constant is subtracted from the instruction register. Otherwise the instruction register is incremented by two, and points to the list element after the branch constant. The sequence of words between BEGIN and UNTIL are thus executed until a non-zero number is left on the data stack at the end of the sequence.

DO . . . LOOP, IF . . . THEN, and other control structures are implemented in a similar manner, using two levels of code: run-time code consisting of system directives, and compile-time code that inserts the appropriate system directives and data into the word definition.

The control word EXECUTE can be executed in the interpreter mode or compiled in a word definition. EXECUTE passes the top entry of the data stack to the address interpreter. Care should be taken to ensure that this number is the compilation address of a valid word.

Conclusion I hope you find, in this example of the implementation of a threaded language, something you can use in solving a programming problem that interests you. Remember that the implementation can (should?) be application-dependent. For the DBMS I mentioned at the beginning of this article, I wrote machine code word definitions for the common disk operations (read a record, write a record, index look-up, etc.) and used high-level definitions to implement the database Find, Create, Delete and other operations. When system development and debugging was complete, I removed the text interpreter and all headers, leaving one word each for the entry point and exit point of the program. The DBMS is thus implemented as a single word.

To get an idea of the speed of this Forth, I coded the Eratosthenes Sieve benchmark (Knuth, 1969; Gilbreath, 1981) and ran it on a Z8002 system at 4MHz. The time for execution of ten iterations of the sieve program was 27 seconds. This compares favorably with with Decus Forth on the PDP 11/70 (11.8 seconds) and polyforth on the TI 990/10 (60.2 seconds).

I would like to thank Jim Gonzalez and Bernard Elliot for their helpful comments on the manuscript.

References:

Gilbreath, J. (1981). "A High-Level Language Benchmark." *Byte 6*:9.

Knuth, D. E. (1969). *The Art of Computer Programming* Vol. 2: Semi-Numerical Algorithms. Addison-Wesley, Reading, MA.

Loeliger, R.G. (1981). *Threaded Interpretive Languages*. Byte Books, Peterborough, NH.

DDJ

(Listing begins on next page)

```
0001 0000 0000             N          NAME FORTH
0002 0000 4D08 00DE R                 CLR BASE                FORCE A START
0003 0004 4D08 00DC R                 CLR START
0004 0008 5E08 01EE R                 JP INIT
0005                                  *
0006                                  *
0007                                  * FORTH LANGUAGE FOR THE Z8000 MICROPROCESSOR
0008                                  *
0009                                  * Copyright (C)  1982 by Louis L. Odette
0010                                  * Permission is granted for individual, personal use in a non-commercial
0011                                  * manner. All commercial rights reserved.
0012                                  *
0013                                  * FORTH is a registered trade mark of FORTH, Inc., 2309 Pacific Coast
0014                                  * Highway, Hermosa Beach CA 90254
0015                                  *
0016                                  ********************
0017                                  *
0018                                  * IN THIS SYSTEM:
0019                                  * R1 IS THE INSTRUCTION REGISTER AND MUST BE PRESERVED BY ALL ROUTINES
0020                                  * R14 IS THE RETURN STACK POINTER
0021                                  * R15 IS THE DATA STACK POINTER
0022                                  *
0023                                  ********************
0024                                  * MACRO DEFINITIONS
0025                                  *
0026                                  HEADER  MACRO
0027                                  * USE: HEADER N,NFIELD,LFIELD
0028                                  * USED TO DEFINE A WORD HEADER IN THE DICTIONARY
0029                                  * N IS THE NUMBER OF CHARACTERS IN THE NAME
0030                                  * NFIELD IS THE FIRST THREE CHARACTERS OF THE NAME
0031                                  * LFIELD IS THE COMPILATION ADDRESS OF THE PREVIOUS ENTRY
0032                                  *
0033                                          DB &0
0034                                          DC '&1'
0035                                          DW &2-6
0036                                          ENDM
0037                                  *
0038                                  *
0039                                  NEXT    MACRO
0040                                  * USE: NEXT IS THE ADDRESS INTERPRETER ROUTINE
0041                                  * THAT TERMINATES ALL MACHINE CODE WORDS
0042                                  *
0043                                          POP R2,@R1
0044                                          JP @R2
0045                                          ENDM
0046                                  *
0047                                  ********************
0048                                  *
0049                                  * HARDWARE DEPENDENT I/O ROUTINES
0050                                  *
0051 000C 5E08 0000        X          EXTERNAL CONOUT
0052 0010 5E08 0000        X          EXTERNAL CONIN
0053                                  *
0054                                  *
0055                                  * The two hardware dependent routines used here are
0056                                  * (1) CONOUT - (output a line to the console) - expects R13 to
0057                                  *              contain the address of the output line, with the output
0058                                  *              line terminated with either 0D(hex) or 04(hex). A 0D terminator
0059                                  *              causes a carriage return-line feed to be sent to the terminal.
0060                                  * (2) CONIN  - (input a line from the console) - gets an input line from the
0061                                  *              console and places it in the input line buffer. CONIN
0062                                  *              terminates the input line with the word 8080(hex), and
0063                                  *              initializes the buffer pointer (BUFPNT in the system data
0064                                  *              area) to the start of the input buffer.
0065                                  *
0066                                  ********************
0067                                  *
0068                                  * DATA AREAS
0069                                  *
0070                                  ********************
0071                                  *
0072                                  * I/O BUFFERS
0073                                  *
0074 0014 0064                        INBUF   DS 100          INPUT BUFFER
0075 0078 0064                        OUTBUF  DS 100          OUTPUT BUFFER
0076                                  *
0077                                  ********************
0078                                  *
0079                                  * SYSTEM DATA AREA: DO NOT CHANGE ORDER OF PARAMETERS
0080                                  *
0081 00DC 0002                        START   DS 2
0082 00DE 0002                        BASE    DS 2
0083 00E0 00                          MODE    DB 0            FLAG
0084 00E1 00                          STATE   DB 0            FLAG
0085 00E2 0002                        DICPNT  DS 2            DICTIONARY POINTER
0086 00E4 0002                        CONXT   DS 2            ADDRESS OF VOCABULARY FOR SEARCH
0087 00E6 0002                        CURRNT  DS 2            ADDRESS OF VOCABULARY FOR COMPILATION
0088 00E8 0002                        BUFPNT  DS 2            ADDRESS OF NEXT WORD TO BE SCANNED
0089 00EA 0002                        STKPNT  DS 2            INITIAL ADDRESS OF SYSTEM STACK
0090 00EC 0A9A             R CMPLER   DW SEMI1-6      ADDRESS OF LAST ENTRY IN COMPILER VOCABULARY
0091                                  *
0092 00EE 0000 00DC        R SYSTEM   EQU START
0093 00EE 0000 0080                   TERM    EQU %80         STRING TERMINATOR
0094                                  ********************
0095                                  *
```

```
0096                            * SYSTEM MESSAGES
0097 00EE 09                    RSTMSG  DB 9
0098 00EF 52                            DC 'RESTART!'
0099 00F7 0D                            DB %0D
0100 00F8 2D                    SRTMSG  DB 45
0101 00F9 5A                            DC 'Z8000-FORTH: COPYRIGHT (C) 1982, L.L. ODETTE'
0102 0125 0D                            DB %0D
0103 0126 03                    OK      DB 3
0104 0127 4F                            DC 'OK'
0105 0129 0D                            DB %0D
0106 012A 02                    NOTKNO  DB 2
0107 012B 3F                            DC '?'
0108 012C 0D                            DB %0D
0109 012D 10                    STKMSG  DB 16
0110 012E 53                            DC 'STACK UNDERFLOW'
0111 013D 0D                            DB %0D
0112                            ***************************
0113                            *
0114                            * ADDRESS INTERPRETER
0115                            *
0116 013E 93E1                  COLON   PUSH @R14,R1            PUSH INTRUCTION REGISTER
0117 0140 97F1                          POP R1,@R15             GET NEW INSTRUCTION
0118                                    NEXT                    DO NEXT
0121                            *
0122 0146 97E1                  SEMI    POP R1,@R14             GET INSTRUCTION FROM RETURN STACK
0123                                    NEXT
0126                            *
0127                            ***************************
0128                            * TEXT INTERPRETER
0129                            *
0130 014C 0750                R OUTINT  DW TYPE                 SEND MESSAGE TO CONSOL
0131 014E 07AC                R         DW QUERY                GET AN INPUT LINE
0132 0150 07F0                R         DW ASPACE
0133 0152 0760                R         DW WORD                 GET THE NEXT WORD IN THE INPUT STREAM
0134 0154 0686                R         DW DROP                 DROP POINTER TO LENGTH BYTE
0135 0156 016E                R         DW SERCH1               LOOK FOR A MATCHING WORD IN THE DICTIONARY
0136 0158 040C                R         DW SYSIF                CONDITIONAL JUMP TO EXECUTE
0137 015A 000E                          DW 14
0138 015C 01C6                R         DW NUMBR1               IF NOT FOUND SEE IF WORD IS A NUMBER
0139 015E 0420                R         DW SYSEND               CONDITIONAL JUMP TO ASPACE
0140 0160 0010                          DW 16
0141 0162 02CA                R         DW QUESTN               IF NOT FOUND AND NOT NUMBER, ECHO WITH ?
0142 0164 042C                R         DW SYSWHL               UNCONDITIONAL JUMP TO TYPE
0143 0166 001A                          DW 26
0144 0168 01A4                R         DW EXEC1                EXECUTE OR COMPILE WORD
0145 016A 042C                R         DW SYSWHL               UNCONDITIONAL JUMP TO ASPACE
0146 016C 001C                          DW 28
0147                              *
0148                              * WORDS USED IN TEXT INTERPRETER
0149                              *
0150 016E 5F00 013E R           SERCH1  CALL COLON              SEARCH ROUTINE
0151 0172 0856                R         DW CONTXT               GET ADDRESS OF CONTEXT VOCABULARY
0152 0174 0474                R         DW FETCH
0153 0176 0474                R         DW FETCH
0154 0178 0368                R         DW SEARCH               SEARCH VOCAB FOR HEADER MATCH
0155 017A 067A                R         DW DUP                  DUPLICATE FLAG
0156 017C 040C                R         DW SYSIF                "IF"
0157 017E 0024                          DW 36                   JUMP TO SEMI IF FALSE (SEARCH SUCCESSFUL)
0158 0180 02B2                R         DW MOD                  OTHERWISE SEE IF WE'RE IN COMPILE MODE
0159 0182 0484                R         DW CFETCH
0160 0184 040C                R         DW SYSIF                "IF"
0161 0186 001C                          DW 28                   JUMP TO SEMI IF IN EXECUTE MODE
0162 0188 0686                R         DW DROP                 OTHERWISE DROP FLAG AND SEARCH COMPILER VOCAB
0163 018A 02C4                R         DW CMPLR                GET ADDRESS OF COMPILER VOCAB
0164 018C 0474                R         DW FETCH
0165 018E 0368                R         DW SEARCH               SEARCH FOR HEADER MATCH
0166 0190 067A                R         DW DUP                  DUPLICATE FLAG
0167 0192 040C                R         DW SYSIF                "IF"
0168 0194 0008                          DW 8                    SET STATE FLAG ACCORDING TO SEARCH FLAG
0169 0196 07C6                R         DW ZERO                 SET TO ZERO IF SEARCH SUCCESSFUL
0170 0198 0418                R         DW SYSELS               "ELSE"
0171 019A 0004                          DW 4
0172 019C 07D4                R         DW ONE                  SET TO ONE IF SEARCH UNSUCCESSFUL
0173 019E 092C                R         DW ST8
0174 01A0 04A4                R         DW CSTORE
0175 01A2 0146                R         DW SEMI
0176                              *
0177 01A4 5F00 013E R           EXEC1   CALL COLON              EXECUTE ROUTINE
0178 01A8 092C                R         DW ST8                  GET SYSTEM STATE FLAG
0179 01AA 0484                R         DW CFETCH
0180 01AC 092C                R         DW ST8                  SET STATE FLAG TO ZERO
0181 01AE 0838                R         DW C0SET
0182 01B0 02B2                R         DW MOD                  GET SYSTEM MODE FLAG
0183 01B2 0484                R         DW CFETCH
0184 01B4 0534                R         DW EQUALS               MODE = STATE ?
0185 01B6 040C                R         DW SYSIF                "IF"
0186 01B8 000A                          DW 10
0187 01BA 0946                R         DW EXECUT               IF EQUAL, EXECUTE COMP ADDR ON STACK
0188 01BC 02F0                R         DW STACK                TEST FOR STACK OVERFLOW
0189 01BE 0418                R         DW SYSELS               "ELSE"
0190 01C0 0004                          DW 4
0191 01C2 08DA                R         DW COMMA                IF NOT EQUAL, ENCLOSE TOP OF STACK IN DICT
0192 01C4 0146                R         DW SEMI
0193                              *
0194 01C6 5F00 013E R           NUMBR1  CALL COLON              NUMBER HANDLING ROUTINE
0195 01CA 0304                R         DW NUMBER               TRY TO CONVERT WORD AT END OF DICT TO A NUMBER
0196 01CC 040C                R         DW SYSIF                "IF"
0197 01CE 0018                          DW 24
0198 01D0 02B2                R         DW MOD                  IF SUCCESSFUL TEST SYSTEM MODE
0199 01D2 0484                R         DW CFETCH
```

(Continued on next page)

```
0200 01D4 040C              R          DW SYSIF                "IF"
0201 01D6 000A                         DW 10
0202 01D8 0434              R          DW LITERL               IF COMPILE, ADD LITERL+NUMB TO DICTIONARY
0203 01DA 0434              R          DW LITERL
0204 01DC 08DA              R          DW COMMA
0205 01DE 08DA              R          DW COMMA
0206 01E0 07C6              R          DW ZERO                 PUSH FALSE FLAG IF CONVERSION SUCCESSFUL
0207 01E2 0418              R          DW SYSELS               "ELSE"
0208 01E4 0004                         DW 4
0209 01E6 07D4              R          DW ONE                  PUSH TRUE FLAG IF CONVERSION UNSUCCESSFUL
0210 01E8 0146              R          DW SEMI
0211                                *
0212                                ************************
0213                                * SYSTEM SUBROUTINES
0214                                *
0215                                * START/RESTART
0216                                *
0217 01EA 0002                      CONVOC  DS 2                STORAGE FOR POINTER TO CONTEXT VOCAB
0218 01EC 0002                      CURVOC  DS 2                STORAGE FOR POINTER TO CURRENT VOCAB
0219                                *
0220 01EE 7602 00EE R               INIT    LDA R2,RSTMSG       LOAD ADDRESS OF RESTART MESSAGE
0221 01F2 4D04 00DE R                       TEST BASE           TEST BASE
0222 01F6 EE1B                              JR NZ,ABORT         START IF ZERO
0223                                *
0224 01F8 210F FFF0                         LD R15,#%FFF0       SET DATA STACK ADDRESS
0225 01FC 6F0F 00EA R                       LD STKPNT,R15       STORE STACK ADDRESS
0226 0200 760C 0AB0 R                       LDA R12,DP          GET FREE MEMORY ADDRESS
0227 0204 6F0C 00E2 R                       LD DICPNT,R12       STORE IN DICTIONARY POINTER
0228 0208 760C 09C0 R                       LDA R12,COLON1-6    GET ADDRESS OF LAST DICTIONARY ENTRY
0229 020C 6F0C 01EA R                       LD CONVOC,R12       STORE IN CONTEXT VOCABULARY
0230 0210 6F0C 01EC R                       LD CURVOC,R12       STORE IN CURRENT VOCABULARY ALSO
0231 0214 760C 01EA R                       LDA R12,CONVOC      GET ADDRESS OF CONTEXT VOCAB
0232 0218 6F0C 00E4 R                       LD CONXT,R12        STORE IN CONXT
0233 021C 760C 01EC R                       LDA R12,CURVOC      GET ADDRESS OF CURRENT VOCAB
0234 0220 6F0C 00E6 R                       LD CURRNT,R12       STORE IN CURRNT
0235 0224 4D05 00DE R 0010                  LD BASE,#%10        MAKE HEX
0236 022A 7602 00F8 R                       LDA R2,SRTMSG       LOAD START MESSAGE
0237                                *
0238 022E 93F2                      ABORT   PUSH @R15,R2        PUSH ADDRESS ON STACK
0239 0230 4D08 00E0 R                       CLR MODE            CLEAR MODE AND STATE
0240 0234 210E FF00                         LD R14,#%FF00       SET RETURN STACK
0241 0238 7601 014C R                       LDA R1,OUTINT       SET INSTRUCT REG TO FIRST WORD OF TEXT INTERP
0242                                        NEXT
0245                                *
0246                                * CONWRT
0247                                *
0248 0240 8C68                      CONWRT  CLRB RH6            EXPECTS MESSAGE ADDRESS IN R12
0249 0242 20CE                              LDB RL6,@R12        GET COUNT
0250 0244 A9C0                              INC R12             BUMP POINTER
0251 0246 760D 0078 R                       LDA R13,OUTBUF      GET BUFFER ADDRESS
0252 024A BAC1 06D0                         LDIRB @R13,@R12,R6  TRANSFER MESSAGE TO BUFFER
0253 024E 0CD5 0404                         LDB @R13,#%04       SET END OF STRING
0254 0252 760D 0078 R                       LDA R13,OUTBUF
0255 0256 5F00 000C R                       CALL CONOUT         PRINT IT
0256 025A 9E08                              RET
0257                                *
0258                                * CRLF
0259                                *
0260 025C 760D 0078 R               CRLF    LDA R13,OUTBUF      GET BUFFER ADDRESS
0261 0260 0CD5 0D0D                         LDB @R13,#%0D       CRLF
0262 0264 5F00 000C R                       CALL CONOUT
0263 0268 9E08                              RET
0264                                *
0265                                * PATCH
0266                                *
0267 026A 4C04 00E0 R               PATCH   TESTB MODE          MODE=0 (EXECUTE)
0268 026E 5E06 022E R                       JP EQ,ABORT         YES
0269 0272 6108 00E6 R                       LD R8,CURRNT        GET ADDRESS OF CURRENT VOCAB
0270 0276 2189                              LD R9,@R8           RESET DICTIONARY POINTER
0271 0278 6F09 00E2 R                       LD DICPNT,R9
0272 027C A993                              INC R9,#4           GET LINK ADDRESS
0273 027E 2199                              LD R9,@R9
0274 0280 2F89                              LD @R8,R9
0275 0282 5E08 022E R                       JP ABORT
0276                                *
0277                                * ASCWD
0278                                *
0279 0286 0DF9 0020                 ASCWD   PUSH @R15,#%0020    PUSH SPACE ON STACK
0280 028A 8D68                      ASCII   CLR R6              CLEAR HIGH WORD
0281 028C 5B06 00DE R                       DIV RR6,BASE        DIVIDE BY BASE
0282 0290 0A0E 0A0A                         CPB RL6,#10         REMAINDER < 10
0283 0294 E102                              JR LT,ASCII0        YES
0284 0296 000E 0707                         ADDB RL6,#7         OTHERWISE ADD OFFSET TO ALPHABET
0285 029A 000E 3030                 ASCII0  ADDB RL6,#'0'       CONVERT TO ASCII
0286 029E 93F6                              PUSH @R15,R6        PUSH ON STACK
0287 02A0 8D74                              TEST R7             QUOTIENT = 0 ?
0288 02A2 EEF3                              JR NE,ASCII         NO
0289 02A4 97F6                      ASCII1  POP R6,@R15         RETIEVE CHARACTERS
0290 02A6 2EDE                              LDB @R13,RL6        PUT CHAR IN BUFFER
0291 02A8 A9D0                              INC R13             BUMP POINTER
0292 02AA 0A0E 2020                         CPB RL6,#%20        WAS CHARACTER A SPACE
0293 02AE EEFA                              JR NE,ASCII1        NO
0294 02B0 9E08                              RET
0295                                *
0296                                ********************
```

```
0297                        *
0298                        * SYSTEM WORDS (HEADERLESS)
0299                        *
0300                        **********
0301                        *
0302                        * SYSTEM VARIABLES
0303                        *
0304 02B2 5F00 0460 R      MOD     CALL SYS                MODE
0305 02B6 0004                     DW 4
0306                        *
0307 02B8 5F00 0460 R      DPNT    CALL SYS                DICTIONARY POINTER
0308 02BC 0006                     DW 6
0309                        *
0310 02BE 5F00 0460 R      BFPNT   CALL SYS                BUFFER POINTER
0311 02C2 000C                     DW 12
0312                        *
0313 02C4 5F00 0460 R      CMPLR   CALL SYS                COMPILER
0314 02C8 0010                     DW 16
0315                        *
0316                        **********
0317                        *
0318                        * SYSTEM I/O, NUMBER CONVERSION, SEARCH
0319                        *
0320 02CA 610C 00E2 R      QUESTN  LD R12,DICPNT           ?QUESTION
0321 02CE A9C0                     INC R12                 GET DICT POINTER AND STEP OVER LENGTH
0322 02D0 26C7                     BITB @R12,#7
0323 02D2 E605                     JR EQ,$+12              IF BIT SET - WORD IS A TERMINATOR
0324 02D4 7602 0126 R              LDA R2,OK               POINT TO "OK" MESSAGE
0325 02D8 93F2                     PUSH @R15,R2            PUT MESSAGE ON STACK
0326                               NEXT
0329 02DE 5F00 025C R              CALL CRLF
0330 02E2 ABC0                     DEC R12                  BACK UP TO WORD LENGTH
0331 02E4 5F00 0240 R              CALL CONWRT             ECHO UNKNOWN WORD
0332 02E8 7602 012A R              LDA R2,NOTKNO           POINT TO UNKNOWN-WORD MESSAGE
0333 02EC 5E08 026A R              JP PATCH
0334                        *
0335 02F0 4B0F 00EA R      STACK   CP R15,STKPNT           COMPARE STKPNT
0336 02F4 5E02 0148 R              JP LE,SEMI+2             NO UNDERFLOW
0337 02F8 610F 00EA R              LD R15,STKPNT            RESET STACK
0338 02FC 7602 012D R              LDA R2,STKMSG           STACK MESSAGE TO WA
0339 0300 5E08 026A R              JP PATCH
0340                        *
0341 0304 8D68             NUMBER  CLR R6                  ?NUMBER
0342 0306 8DA8                     CLR R10
0343 0308 6108 00E2 R              LD R8,DICPNT            GET DICTIONARY POINTER
0344 030C 2080                     LDB RH0,@R8             GET COUNT
0345 030E A980                     INC R8                  STEP OVER COUNT
0346                        *
0347 0310 0C81 2D2D                CPB @R8,#'-'             CHECK FOR MINUS SIGN
0348 0314 8C88                     CLRB RL0                SET SIGN FLAG FALSE
0349 0316 EE03                     JR NE,$+8               SKIP IF NOT '-'
0350 0318 AA80                     DECB RL0                SET FLAG TRUE
0351 031A AA00                     DECB RH0                DECREASE COUNT
0352 031C A980                     INC R8                  NEXT CHAR
0353                        *
0354 031E 0DF9 0000                PUSH @R15,#0            SET 0 RESULT
0355 0322 208E             NLOOP   LDB RL6,@R8             GET CHAR
0356 0324 020E 3030                SUBB RL6,#%30           SUBTRACT NUMBER OFFSET
0357 0328 E50B                     JR MI,NOTNO             NOT NUMBER IF <%30
0358 032A 0A0E 0A0A                CPB RL6,#10
0359 032E E505                     JR MI,NUMB              IF <10 ITS A DIGIT
0360 0330 0A0E 1111                CPB RL6,#17
0361 0334 E505                     JR MI,NOTNO             NOT A NUMBER IF <17 AND >10
0362 0336 020E 0707                SUBB RL6,#7             SUBTRACT OFFSET
0363                        *
0364 033A 4B06 00DE R      NUMB    CP R6,BASE              COMPARE WITH THE BASE
0365 033E E504                     JR MI,$+10               OK IF <BASE
0366 0340 0DF5 0000        NOTNO   LD @R15,#0              SET FALSE FLAG
0367                               NEXT
0370                        *
0371 0348 21FB                     LD R11,@R15             GET RESULT
0372 034A 590A 00DE R              MULT RR10,BASE           MULTIPLY BY BASE
0373 034E 816B                     ADD R11,R6              ADD NUM TO LOW ORDER WORD
0374 0350 2FFB                     LD @R15,R11             SAVE RESULT
0375                        *
0376 0352 A980                     INC R8                  NEXT CHAR
0377 0354 F01A                     DBJNZ RH0,NLOOP         DECREMENT COUNT AND LOOP
0378                        *
0379 0356 8C84                     TESTB RL0               TEST SIGN WHEN FINISHED
0380 0358 E603                     JR EQ,$+8               PLUS
0381 035A 8DB8                     CLR R11                 MINUS
0382 035C 03FB                     SUB R11,@R15            COMPLEMENT
0383 035E 2FFB                     LD @R15,R11
0384                        *
0385 0360 0DF9 00FF                PUSH @R15,#%FF          TRUE FLAG
0386                               NEXT
0389                        *
0390 0368 21F8             SEARCH  LD R8,@R15
0391 036A 6109 00E2 R              LD R9,DICPNT            GET DICTIONARY POINTER
0392 036E 2090                     LDB RH0,@R9             GET CHAR COUNT
0393 0370 0A80                     CPB RH0,@R8             SAME AS IN FIRST HEADER?
0394 0372 EE0E                     JR NE,NXTHDR            NO? - GET NEXT HEADER
0395                        *
0396 0374 0A00 0404                CPB RH0,#4              LENGTH>4
0397 0378 E101                     JR LT,NXTCH
0398 037A C003                     LDB RH0,#3              NO - SET TO 3
0399                        *
0400 037C A990             NXTCH   INC R9
0401 037E A980                     INC R8                  BUMP POINTERS
0402 0380 2098                     LDB RL0,@R9             GET NEXT CHAR
```

(Continued on next page)

```
0403 0382 0A88                          CPB RL0,@R8             COMPARE
0404 0384 EE05                          JR NE,NXTHDR
0405 0386 F006                          DBJNZ RH0,NXTCH
0406                            *
0407 0388 29F5                          INC @R15,#6             IF MATCH POINT TO CODE ADDRESS
0408 038A 0DF9 0000                     PUSH @R15,#0            FLAG = 0
0409 038E E808                          JR $+18
0410                            *
0411 0390 21F8                  NXTHDR  LD R8,@R15
0412 0392 A983                          INC R8,#4               POINT TO LINK
0413 0394 2187                          LD R7,@R8               GET IT
0414 0396 2FF7                          LD @R15,R7
0415 0398 8D74                          TEST R7                 TEST LINK
0416 039A EEE6                          JR NE,SEARCH            IF NOT END - DO MORE SEARCH
0417                            *
0418 039C 0DF5 0001                     LD @R15,#1              SET FLAG=1 IF END OF DICTIONARY
0419                                    NEXT                    DO NEXT
0422                            *
0423 03A4 5F00 013E R           ENDC    CALL COLON
0424 03A8 08DA               R          DW COMMA                STACK ON ENTRY: CNTRL ADDRS,JUMP ADDR
0425 03AA 0938               R          DW HERE
0426 03AC 0692               R          DW SWAP
0427 03AE 0588               R          DW MINUS
0428 03B0 08DA               R          DW COMMA
0429 03B2 0146               R          DW SEMI                 STACK ON EXIT: NULL
0430                            *
0431 03B4 5F00 013E R           DOC     CALL COLON
0432 03B8 08DA               R          DW COMMA                ENCLOSE ADDRESS OF SYSDO DIRECTIVE
0433 03BA 0938               R          DW HERE                 LEAVE ADDRESS OF NEXT FREE DICTIONARY ENTRY
0434 03BC 0146               R          DW SEMI
0435                            *
0436 03BE 5F00 013E R           BUILD   CALL COLON              MAKE A HEADER IN THE DICTIONARY
0437 03C2 03E6               R          DW ENTRY
0438 03C4 07F0               R          DW ASPACE
0439 03C6 0760               R          DW WORD
0440 03C8 0862               R          DW CRNT
0441 03CA 0474               R          DW FETCH                GET CURRENT VOCAB
0442 03CC 0496               R          DW STORE                UPDATE IT
0443 03CE 0434               R          DW LITERL
0444 03D0 0004                          DW 4
0445 03D2 02B8               R          DW DPNT
0446 03D4 04B2               R          DW PSTORE               UPDATE DICTIONARY
0447 03D6 08DA               R          DW COMMA                ENCLOSE LINK TO LAST ENTRY
0448 03D8 0434               R          DW LITERL
0449 03DA 5F00                          DW %5F00                PUT A CALL INSTRUCTION AT COMPILATION ADDRESS
0450 03DC 08DA               R          DW COMMA
0451 03DE 0434               R          DW LITERL
0452 03E0 013E               R          DW COLON
0453 03E2 08DA               R          DW COMMA                ARGUEMENT OF CALL IS COLON
0454 03E4 0146               R          DW SEMI
0455                            *
0456 03E6 5F00 013E R           ENTRY   CALL COLON
0457 03EA 0862               R          DW CRNT
0458 03EC 0474               R          DW FETCH
0459 03EE 0474               R          DW FETCH                LEAVE ADDRESS OF CURRENT VOCABULARY ENTRY
0460 03F0 0146               R          DW SEMI
0461                            *
0462 03F2 5F00 013E R           CAXCLM  CALL COLON
0463 03F6 03E6               R          DW ENTRY
0464 03F8 0434               R          DW LITERL
0465 03FA 0008                          DW 8
0466 03FC 0578               R          DW PLUS                 COMPUTE ADDRESS OF CALL ARGUEMENT OF ENTRY
0467 03FE 0496               R          DW STORE                REPLACE WITH TOP OF STACK
0468 0400 0146               R          DW SEMI
0469                            *
0470 0402 5F00 013E R           SCODE   CALL COLON
0471 0406 0652               R          DW RFROM
0472 0408 03F2               R          DW CAXCLM               PUT RETURN STACK ENTRY IN CALL ARGUEMENT
0473 040A 0146               R          DW SEMI
0474                            *
0475                            **********
0476                            *
0477                            * SYSTEM DIRECTIVES: USED TO IMPLEMENT CONTROL STRUCTURES
0478                            *
0479 040C 97F8                  SYSIF    POP R8,@R15            *IF
0480 040E 8D84                           TEST R8                TEST TOP OF STACK
0481 0410 E603                           JR EQ,SYSELS             DO RELATIVE JUMP
0482 0412 A911                          INC R1,#2               OTHERWISE BUMP IR
0483                                    NEXT
0486                            *
0487 0418 2116                  SYSELS  LD R6,@R1               *ELSE
0488 041A 8161                          ADD R1,R6               BUMP INTRUCTION REG
0489                                    NEXT
0492                            *
0493 0420 97F8                  SYSEND  POP R8,@R15             *END
0494 0422 8D84                          TEST R8                 TEST FLAG
0495 0424 E603                          JR EQ,SYSWHL             DO RELATIVE JUMP
0496 0426 A911                          INC R1,#2               OTHERWISE BUMP IR
0497                                    NEXT
0500                            *
0501 042C 2116                  SYSWHL  LD R6,@R1               *WHILE
0502 042E 8361                          SUB R1,R6               BUMP INTRUCTION REG
0503                                    NEXT
0506                            *
0507 0434 13F1                  LITERL  PUSH @R15,@R1           *#
```

```
0508 0436 A911                          INC R1,#2               BUMP INSTRUCTION REG
0509                                    NEXT
0512                            *
0513 043C 97E8                  SYSPLP  POP R8,@R14             *+LOOP
0514 043E 97F7                          POP R7,@R15             GET INCREMENT FROM STACK
0515 0440 E803                          JR LP
0516                            *
0517 0442 97E8                  SYSLOP  POP R8,@R14             *LOOP
0518 0444 2107 0001                     LD R7,#1                UNIT
0519 0448 8178                  LP      ADD R8,R7
0520 044A 0BE8                          CP R8,@R14              COMPARE INDEX WITH TERMINAL COUNT
0521 044C 93E8                          PUSH @R14,R8            RESTORE INDEX
0522 044E E5EE                          JR MI,SYSWHL            DO RELATIVE JUMP WHILE MINUS
0523 0450 95E8                          POPL RR8,@R14           ELSE CLEAN STACK
0524 0452 A911                          INC R1,#2               BUMP INSTRUCTION REG (STEP OVER JUMP)
0525                                    NEXT
0528                            *
0529 0458 95F8                  SYSDO   POPL RR8,@R15           *DO
0530 045A 91E8                          PUSHL @R14,RR8          MOVE INDEX AND TERMINAL COUNT TO RETURN STACK
0531                                    NEXT
0534                            *
0535 0460 7608 00DC R           SYS     LDA R8,SYSTEM           *SYS
0536 0464 97F2                          POP R2,@R15             POP ADDRESS OF OFFSET
0537 0466 0128                          ADD R8,@R2              ADD OFFSET
0538 0468 93F8                          PUSH @R15,R8            PUSH ADDRESS ON STACK
0539                                    NEXT
0542                            *
0543                            ********************
0544                            *
0545                            * DICTIONARY WORD SETS
0546                            *
0547                            ********************
0548                            *
0549                            * MEMORY REFERENCE WORD SET
0550                            *
0551                                    HEADER 1,@XX,6,         @
0555 0474 21F6                  FETCH   LD R6,@R15
0556 0476 2166                          LD R6,@R6               REPLACE ADDRESS AT TOP OF DATA STACK

0557 0478 2FF6                          LD @R15,R6              WITH WORD AT THAT ADDRESS
0558                                    NEXT
0561                            *
0562                                    HEADER 2,C@X,FETCH,     C@
0566 0484 21F6                  CFETCH  LD R6,@R15
0567 0486 206D                          LDB RL5,@R6             REPLACE ADDRESS AT TOP OF DATA STACK
0568 0488 B1D0                          EXTSB RL5               WITH BYTE AT THAT ADDRESS, SIGN EXTENDED
0569 048A 2FF5                          LD @R15,R5              TO HIGH BYTE OF STACK WORD
0570                                    NEXT
0573                            *
0574                                    HEADER 1,!XX,CFETCH,    !
0578 0496 95F6                  STORE   POPL RR6,@R15
0579 0498 2F67                          LD @R6,R7               STORE TOP-1 AT TOP ADDRESS
0580                                    NEXT
0583                            *
0584                                    HEADER 2,C!X,STORE,     C!
0588 04A4 95F6                  CSTORE  POPL RR6,@R15
0589 04A6 2E6F                          LDB @R6,RL7             STORE LOW BYTE OF TOP-1 AT TOP ADDRESS
0590                                    NEXT
0593                            *
0594                                    HEADER 2,+!X,CSTORE,    +!
0598 04B2 95F6                  PSTORE  POPL RR6,@R15
0599 04B4 0167                          ADD R7,@R6              ADD TOP-1 TO WORD AT TOP
0600 04B6 2F67                          LD @R6,R7
0601                                    NEXT
0604                            *
0605                                    HEADER 1,?XX,PSTORE,    ?
0609 04C2 97F6                  QMARK   POP R6,@R15
0610 04C4 2167                          LD R7,@R6               GET WORD AT ADDRESS ON DATA STACK
0611 04C6 760D 0078 R                   LDA R13,OUTBUF          POINT TO BUFFER
0612 04CA 5F00 0286 R                   CALL ASCWD              CONVERT NUMBER TO ASCII
0613 04CE 0CD5 0404                     LDB @R13,#$04           END OF LINE
0614 04D2 760D 0078 R                   LDA R13,OUTBUF
0615 04D6 5F00 000C R                   CALL CONOUT             PRINT CONVERTED NUMBER
0616                                    NEXT
0619                            *
0620                                    HEADER 4,MOV,QMARK,     MOVE
0624 04E4 95F6                  MOVE    POPL RR6,@R15           GET COUNT AND DESTINATION ADDRESS
0625 04E6 97F5                          POP R5,@R15             GET SOURCE ADDRESS
0626 04E8 A76F                          BIT R6,#15              TEST SIGN BIT
0627 04EA EE02                          JR NE,$+6               MOVE WORDS IF COUNT IS POSITIVE
0628 04EC BB51 0670                     LDIR @R7,@R5,R6
0629                                    NEXT
0632                            *
0633                                    HEADER 5,CMO,MOVE,      CMOVE
0637 04FA 95F6                  CMOVE   POPL RR6,@R15           GET COUNT AND DESTINATION ADDRESS
0638 04FC 97F5                          POP R5,@R15             GET SOURCE ADDRESS
0639 04FE A76F                          BIT R6,#15
0640 0500 EE02                          JR NE,$+6               MOVE BYTES OF COUNT IS POSITIVE
0641 0502 BA51 0670                     LDIRB @R7,@R5,R6
0642                                    NEXT
0645                            *
0646                                    HEADER 4,FIL,CMOVE,     FILL
0650 0510 95F6                  FILL    POPL RR6,@R15           GET FILL BYTE AND COUNT
0651 0512 97F5                          POP R5,@R15             GET START ADDRESS

0652 0514 A77F                          BIT R7,#15
0653 0516 EE03                          JR NE,$+8
0654 0518 2E5E                          LDB @R5,RL6             STORE BYTE
0655 051A A950                          INC R5
0656 051C F783                          DJNZ R7,$-4
```

(Continued on next page)

```
0657                                    NEXT
0660                            *
0661                            ********************
0662                            *
0663                            * COMPARISON WORD SET
0664                            *
0665                                    HEADER 3,NOT,FILL,      NOT
0669 0528 0DF0                  NOT     COM @R15                COMPLEMENT NUMBER ON STACK
0670                                    NEXT
0673                            *
0674                                    HEADER 1,=XX,NOT,       =
0678 0534 97F6                  EQUALS  POP R6,@R15
0679 0536 0BF6                          CP R6,@R15              COMPARE TOP WITH TOP-1
0680 0538 0DF8                          CLR @R15                SET FALSE FLAG
0681 053A EE01                          JR NE,$+4
0682 053C 0DF0                          COM @R15                SET TRUE IF THEY'RE EQUAL
0683                                    NEXT
0686                            *
0687                                    HEADER 2,0=X,EQUALS,    0=
0691 0548 E8EF                  EQUAL0  JR NOT                  COMPLEMENT TOP
0692                            *
0693                                    HEADER 1,<XX,EQUAL0,    <
0697 0550 97F6                  LESS    POP R6,@R15
0698 0552 0BF6                          CP R6,@R15              COMPARE TOP WITH TOP-1
0699 0554 0DF8                          CLR @R15                SET FALSE FLAG
0700 0556 E201                          JR LE,$+4
0701 0558 0DF0                          COM @R15                SET TRUE IF TOP < TOP-1
0702                                    NEXT
0705                            *
0706                                    HEADER 1,>XX,LESS,      >
0710 0564 97F6                  GREATR  POP R6,@R15
0711 0566 0BF6                          CP R6,@R15              COMPARE TOP WITH TOP-1
0712 0568 0DF8                          CLR @R15                SET FALSE FLAG
0713 056A E901                          JR GE,$+4
0714 056C 0DF0                          COM @R15                SET TRUE IF TOP > TOP-1
0715                                    NEXT
0718                            *
0719                            ********************
0720                            *
0721                            * ARITHMETIC AND LOGICAL WORD SET
0722                            *
0723                                    HEADER 1,+XX,GREATR,    +
0727 0578 97F6                  PLUS    POP R6,@R15
0728 057A 01F6                          ADD R6,@R15             ADD TOP TO TOP-1
0729 057C 2FF6                          LD @R15,R6              LEAVE RESULT ON STACK
0730                                    NEXT
0733                            *
0734                                    HEADER 1,-XX,PLUS,      -
0738 0588 95F6                  MINUS   POPL RR6,@R15
0739 058A 8367                          SUB R7,R6               SUBTRACT TOP FROM TOP-1
0740 058C 93F7                          PUSH @R15,R7
0741                                    NEXT
0744                            *
0745                                    HEADER 2,1+X,MINUS,     1+
0749 0598 29F0                  PLUS1   INC @R15                INCREMENT TOP
0750                                    NEXT
0753                            *
0754                                    HEADER 2,1-X,PLUS1,     1-
0758 05A4 2BF0                  MINUS1  DEC @R15                DECREMENT TOP
0759                                    NEXT
0762                            *
0763                                    HEADER 2,2+X,MINUS1,    2+
0767 05B0 29F1                  PLUS2   INC @R15,#2             INCREMENT TOP BY 2
0768                                    NEXT
0771                            *
0772                                    HEADER 2,2-X,PLUS2,     2-
0776 05BC 2BF1                  MINUS2  DEC @R15,#2             DECREMENT TOP BY 2
0777                                    NEXT
0780                            *
0781                                    HEADER 1,*XX,MINUS2,    *
0785 05C8 95F6                  TIMES   POPL RR6,@R15
0786 05CA A165                          LD R5,R6
0787 05CC 9956                          MULT RR6,R5             MULTIPLY TOP BY TOP-1
0788 05CE 93F7                          PUSH @R15,R7            LEAVE RESULT ON STACK
0789                                    NEXT
0792                            *
0793                                    HEADER 1,/XX,TIMES,     /
0797 05DA 95F6                  DIVIDE  POPL RR6,@R15
0798 05DC A165                          LD R5,R6
0799 05DE 8D68                          CLR R6
0800 05E0 9B56                          DIV RR6,R5              DIVIDE TOP-1 BY TOP
0801 05E2 93F7                          PUSH @R15,R7            LEAVE QUOTIENT ON STACK
0802                                    NEXT
0805                            *
0806                                    HEADER 3,MOD,DIVIDE,    MOD
0810 05EE 95F6                  MOD0    POPL RR6,@R15
0811 05F0 A165                          LD R5,R6
0812 05F2 8D68                          CLR R6
0813 05F4 9B56                          DIV RR6,R5              DIVIDE TOP-1 BY TOP
0814 05F6 93F6                          PUSH @R15,R6            LEAVE REMAINDER
0815                                    NEXT
0818                            *
0819                                    HEADER 4,/MO,MOD0,      /MOD
0823 0602 95F6                  DIVMOD  POPL RR6,@R15
0824 0604 A165                          LD R5,R6
```

```
0825 0606 8D68                          CLR R6
0826 0608 9B56                          DIV RR6,R5              DIVIDE TOP-1 BY TOP
0827 060A AD76                          EX R6,R7
0828 060C 91F6                          PUSHL @R15,RR6          LEAVE QUOTIENT,REMAINDER ON STACK
0829                                    NEXT
0832                            *
0833                                    HEADER 6,NEG,DIVMOD,    NEGATE
0837 0618 0DF2                  NEGATE  NEG @R15                 REPLACE TOP WITH ITS 2'S COMPLEMENT
0838                                    NEXT
0841                            *
0842                                    HEADER 3,ABS,NEGATE,    ABS
0846 0624 27FF                  ABS     BIT @R15,#15            TOP NEGATIVE?
0847 0626 EEF8                          JR NE,NEGATE            YES - NEGATE
0848                                    NEXT
0851                            *
0852                                    HEADER 3,AND,ABS,       AND
0856 0632 97F6                  AND0    POP R6,@R15
0857 0634 07F6                          AND R6,@R15             AND TOP WITH TOP-1
0858 0636 2FF6                          LD @R15,R6              LEAVE RESULT ON STACK
0859                                    NEXT
0862                            *
0863                                    HEADER 2,ORX,AND0,      OR
0867 0642 97F6                  OR0     POP R6,@R15
0868 0644 05F6                          OR R6,@R15              OR TOP AND TOP-1
0869 0646 2FF6                          LD @R15,R6              LEAVE RESULT ON STACK
0870                                    NEXT
0873                            *
0874                            ********************
0875                            *
0876                            * STACK MANIPULATION
0877                            *
0878                            *
0879                                    HEADER 2,R>X,OR0,       R>
0883 0652 97E6                  RFROM   POP R6,@R14
0884 0654 93F6                          PUSH @R15,R6            PUT TOP OF RETURN STACK ON DATA STACK
0885                                    NEXT
0888                            *
0889                                    HEADER 2,>RX,RFROM,     >R
0893 0660 97F6                  TOR     POP R6,@R15
0894 0662 93E6                          PUSH @R14,R6            MOVE TOP OF DATA STACK TO RETURN STACK
0895                                    NEXT
0898                            *
0899                                    HEADER 2,R@X,TOR,       R@
0903 066E 13FE                  RFETCH  PUSH @R15,@R14          COPY TOP OF RETURN STACK TO DATA STACK
0904                                    NEXT
0907                            *
0908                                    HEADER 3,DUP,RFETCH,    DUP
0912 067A 13FF                  DUP     PUSH @R15,@R15          DUPLICATE TOP
0913                                    NEXT
0916                            *
0917                                    HEADER 4,DRO,DUP,       DROP
0921 0686 97F6                  DROP    POP R6,@R15             DROP TOP
0922                                    NEXT
0925                            *
0926                                    HEADER 4,SWA,DROP,      SWAP
0930 0692 97F6                  SWAP    POP R6,@R15
0931 0694 2DF6                          EX R6,@R15              EXCHANGE TWO TOP STACK ENTRIES
0932 0696 93F6                          PUSH @R15,R6
0933                                    NEXT
0936                            *
0937                                    HEADER 4,OVE,SWAP,      OVER
0941 06A2 53FF 0002             OVER    PUSH @R15,2(R15)        COPY TOP-1
0942                                    NEXT
0945                            *
0946                                    HEADER 3,ROT,OVER,      ROT
0950 06B0 95F6                  ROT     POPL RR6,@R15
0951 06B2 97F5                          POP R5,@R15
0952 06B4 91F6                          PUSHL @R15,RR6
0953 06B6 93F5                          PUSH @R15,R5            ROTATE THIRD ITEM TO TOP
0954                                    NEXT
0957                            *
0958                                    HEADER 4,ROL,ROT,       ROLL
0962 06C2 97F4                  ROLL    POP R4,@R15             GET COUNT
0963 06C4 AB40                          DEC R4
0964 06C6 A1F6                          LD R6,R15
0965 06C8 8146                          ADD R6,R4
0966 06CA 8146                          ADD R6,R4               POINT TO WORD TO BE ROTATED
0967 06CC 2167                          LD R7,@R6               SAVE WORD
0968 06CE A165                          LD R5,R6
0969 06D0 AB51                          DEC R5,#2               POINT TO NEXT HIGHER WORD IN STACK
0970 06D2 BB59 0460                     LDDR @R6,@R5,R4         SHIFT STACK
0971 06D6 2FF7                          LD @R15,R7              PUT N'TH ITEM ON TOP
0972                                    NEXT
0975                            *
0976                                    HEADER 4,PIC,ROLL,      PICK
0980 06E2 97F4                  PICK    POP R4,@R15             GET COUNT
0981 06E4 AB40                          DEC R4
0982 06E6 A1F6                          LD R6,R15
0983 06E8 8146                          ADD R6,R4
0984 06EA 8146                          ADD R6,R4               POINT TO WOULD TO BE COPIED
0985 06EC 13F6                          PUSH @R15,@R6           COPY WORD TO TOP
0986                                    NEXT
0989                            *
0990                                    HEADER 4,?DU,PICK,      ?DUP
0994 06F8 0DF4                  QDUP    TEST @R15
0995 06FA E601                          JR EQ,$+4
0996 06FC 13FF                          PUSH @R15,@R15          COPY TOP IF NOT ZERO
0997                                    NEXT
1000                            *
1001                            ********************
```

(Continued on next page)

```
1002                                *
1003                                * TERMINAL INPUT-OUTPUT WORDS
1004                                *
1005                                        HEADER 2,CRX,QDUP,      CR
1009 0708 760D 0078 R               CR      LDA R13,OUTBUF
1010 070C 0CD5 0D0D                         LDB @R13,#%0D
1011 0710 5F00 000C R                       CALL CONOUT             DO A CARRIAGE RETURN (AUTO LINEFEED ASSUMED)
1012                                        NEXT
1015                                *
1016                                        HEADER 4,EMI,CR,        EMIT
1020 071E 760D 0078 R               EMIT    LDA R13,OUTBUF
1021 0722 97F6                              POP R6,@R15
1022 0724 2EDE                              LDB @R13,RL6
1023 0726 4CD5 0001   0404                  LDB 1(R13),#%04         SET END OF STRING
1024 072C 5F00 000C R                       CALL CONOUT
1025                                        NEXT
1028                                *
1029                                        HEADER 5,SPA,EMIT,      SPACE
1033 073A 760D 0078 R               SPACE   LDA R13,OUTBUF
1034 073E 0DD5 2004                         LD @R13,#%2004          SET SPACE AND END-OF-STRING
1035 0742 5F00 000C R                       CALL CONOUT             WRITE IT
1036                                        NEXT
1039                                *
1040                                        HEADER 4,TYP,SPACE,     TYPE
1044 0750 97FC                      TYPE    POP R12,@R15
1045 0752 5F00 0240 R                       CALL CONWRT             GET ADDRESS OF STRING AND WRITE STRING
1046                                        NEXT
1049                                *
1050                                        HEADER 4,WOR,TYPE,      WORD
1054 0760 6108 00E8 R               WORD    LD R8,BUFPNT
1055 0764 6109 00E2 R                       LD R9,DICPNT            GET BUFFER AND DICTIONARY POINTERS
1056 0768 97F6                              POP R6,@R15             GET TERMINATOR FROM STACK
1057 076A 0A0E 2020                         CPB RL6,#%20             SPACE?
1058 076E EE05                              JR NE,TOK               NO - THEN START
1059 0770 0C81 2020                 IGNLB   CPB @R8,#%20            IGNORE SPACES IN BUFFER
1060 0774 EE02                              JR NE,TOK
1061 0776 A980                              INC R8
1062 0778 E8FB                              JR IGNLB
1063 077A 93F8                      TOK     PUSH @R15,R8            SAVE STARTING ADDRESS
1064 077C A860                      COUNT   INCB RH6                KEEP COUNT IN RH6
1065 077E A980                              INC R8
1066 0780 0A8E                              CPB RL6,@R8
1067 0782 E604                              JR EQ,ENDTOK             FIND NEXT SEPARATOR
1068 0784 0C81 8080                         CPB @R8,#TERM           COMPARE WITH TERMINATOR
1069 0788 EEF9                              JR NE,COUNT
1070 078A AB80                              DEC R8
1071 078C A980                      ENDTOK  INC R8                  BUMP POINTER
1072 078E 6F08 00E8 R                       LD BUFPNT,R8            RESET POINTER
1073 0792 2E96                              LDB @R9,RH6             WRITE COUNT AT START OF ENTRY
1074 0794 21F8                              LD R8,@R15              GET STARTING ADDRESS OF WORD IN BUFFER
1075 0796 2FF9                              LD @R15,R9              LEAVE ADDRESS OF LENGTH BYTE ON STACK
1076 0798 A990                              INC R9                  BUMP POINTER
1077 079A A06E                              LDB RL6,RH6
1078 079C 8C68                              CLRB RH6
1079 079E BA81 0690                         LDIRB @R9,@R8,R6        MOVE THE WORD TO THE DICTIONARY
1080                                        NEXT                    DO NEXT
1083                                *
1084                                        HEADER 5,QUE,WORD,      QUERY
1088 07AC 5F00 0010 R               QUERY   CALL CONIN              GET AN INPUT LINE FROM THE CONSOL
1089                                        NEXT
1092                                *
1093                                ********************
1094                                *
1095                                * MISCELLANEOUS WORDS
1096                                *
1097                                        HEADER 4,BAS,QUERY,     BASE
1101 07BA 5F00 0460 R               BASE0   CALL SYS
1102 07BE 0002                              DW 2
1103                                *
1104                                        HEADER 1,0XX,BASE0,     ZERO
1108 07C6 0DF9 0000                 ZERO    PUSH @R15,#0
1109                                        NEXT
1112                                *
1113                                        HEADER 1,1XX,ZERO,      ONE
1117 07D4 0DF9 0001                 ONE     PUSH @R15,#1
1118                                        NEXT
1121                                *
1122                                        HEADER 1,2XX,ONE,       TWO
1126 07E2 0DF9 0002                 TWO     PUSH @R15,#2
1127                                        NEXT
1130                                *
1131                                        HEADER 6,ASP,TWO,       ASPACE
1135 07F0 0DF9 0020                 ASPACE  PUSH @R15,#%20
1136                                        NEXT
1139                                *
1140                                        HEADER 1,.XX,ASPACE,
1144 07FE 97F7                      PERIOD  POP R7,@R15
1145 0800 760D 0078 R                       LDA R13,OUTBUF
1146 0804 5F00 0286 R                       CALL ASCWD              CONVERT WORD AT TOP OF STACK TO ASCII
1147 0808 0CD5 0404                         LDB @R13,#04            SET END-OF-STRING
1148 080C 760D 0078 R                       LDA R13,OUTBUF
1149 0810 5F00 000C R                       CALL CONOUT             PRINT IT
1150                                        NEXT
1153                                *
1154                                        HEADER 4,2DU,PERIOD,    2DUP
```

```
1158 081E 13FF                DUP2    PUSH @R15,@R15
1159 0820 13FF                        PUSH @R15,@R15
1160                                  NEXT
1163                          *
1164                                  HEADER 5,2DR,DUP2,      2DROP
1168 082C 95F6                DROP2   POPL RR6,@R15           DROP TOP TWO ENTRIES
1169                                  NEXT
1172                          *
1173                                  HEADER 5,C0S,DROP2,     C0SET
1177 0838 97F6                C0SET   POP R6,@R15
1178 083A 0C68                        CLRB @R6                CLEAR BYTE AT TOP
1179                                  NEXT
1182                          *
1183                                  HEADER 5,C1S,C0SET,     C1SET
1187 0846 97F6                C1SET   POP R6,@R15
1188 0848 0C65 0101                   LDB @R6,#1              SET BYTE = 1 AT TOP
1189                                  NEXT
1192                          *
1193                          ********************
1194                          *
1195                          * VOCABULARIES
1196                          *
1197                          *
1198                                  HEADER 7,CON,C1SET,     CONTEXT
1202 0856 5F00 0460 R         CONTXT  CALL SYS
1203 085A 0008                        DW 8
1204                          *
1205                                  HEADER 7,CUR,CONTXT,    CURRENT
1209 0862 5F00 0460 R         CRNT    CALL SYS
1210 0866 000A                        DW 10
1211                          *
1212                                  HEADER 11,DEF,CRNT,     DEFINITIONS
1216 086E 5F00 013E R         DEFN    CALL COLON
1217 0872 0862                R       DW CRNT                 SET CURRENT TO CONTEXT
1218 0874 0474                R       DW FETCH
1219 0876 0856                R       DW CONTXT
1220 0878 0496                R       DW STORE
1221 087A 0146                R       DW SEMI
1222                          *
1223                                  HEADER 4,FIN,DEFN,      FIND
1227 0882 5F00 013E R         FIND    CALL COLON
1228 0886 07F0                R       DW ASPACE
1229 0888 0760                R       DW WORD                 READ NEXT WORD FROM INPUT BUFFER
1230 088A 0686                R       DW DROP                 DROP ADDRESS OF WORD
1231 088C 0856                R       DW CONTXT
1232 088E 0474                R       DW FETCH
1233 0890 0474                R       DW FETCH
1234 0892 0368                R       DW SEARCH               SEARCH CONTEXT VOCABULARY FOR WORD
1235 0894 040C                R       DW SYSIF                "IF"
1236 0896 0004                        DW 4
1237 0898 07C6                R       DW ZERO
1238 089A 0146                R       DW SEMI
1239                          *
1240                                  HEADER 6,FOR,FIND,      FORGET
1244 08A2 5F00 013E R         FORGET  CALL COLON
1245 08A6 07F0                R       DW ASPACE
1246 08A8 0760                R       DW WORD                 GET NEXT WORD FROM INPUT STREAM
1247 08AA 0686                R       DW DROP                 DROP ADDRESS
1248 08AC 0862                R       DW CRNT                 GET ADDRESS OF CURRENT VOCABULARY
1249 08AE 0474                R       DW FETCH
1250 08B0 0474                R       DW FETCH
1251 08B2 0368                R       DW SEARCH               SEARCH FOR THE WORD IN DICTIONARY
1252 08B4 040C                R       DW SYSIF                ERROR IF NOT FOUND
1253 08B6 0004                        DW 4
1254 08B8 02CA                R       DW QUESTN
1255 08BA 067A                R       DW DUP                  DUPLICATE COMPILATION ADDRESS
1256 08BC 07E2                R       DW TWO
1257 08BE 0588                R       DW MINUS                SUBTRACT TWO TO GET LINK
1258 08C0 0474                R       DW FETCH
1259 08C2 0862                R       DW CRNT
1260 08C4 0474                R       DW FETCH                GET CURRENT VOCAB
1261 08C6 0496                R       DW STORE                SET CURRENT VOCAB TO LINK OF FORGOTTEN WORD
1262 08C8 0434                R       DW LITERL
1263 08CA 0006                        DW 6
1264 08CC 0588                R       DW MINUS                SUBTRACT 6 TO POINT TO HEADER
1265 08CE 02B8                R       DW DPNT
1266 08D0 0496                R       DW STORE                RESET DICTIONARY POINTER
1267 08D2 0146                R       DW SEMI
1268                          *
1269                          *
1270                          ********************
1271                          *
1272                          *
1273                          * DEFINING WORDS, COMPILER WORDS, CONTROL WORDS
1274                          *
1275                          *
1276 08D4 01                          DB 1
1277 08D5 2C                          DC ',  '
1278 08D8 089C                R       DW FORGET-6
1279 08DA 97F6                COMMA   POP R6,@R15
1280 08DC 6107 00E2 R                 LD R7,DICPNT
1281 08E0 2F76                        LD @R7,R6               PUT TOP OF DATA STACK INTO DICTIONARY
1282 08E2 6901 00E2 R                 INC DICPNT,#2           INCREMENT DICTIONARY POINTER
1283                                  NEXT
1286                          *
1287                                  HEADER 5,ALL,COMMA,     ALLOT
1291 08F0 97F6                ALLOT   POP R6,@R15             GET COUNT
1292 08F2 4106 00E2 R                 ADD R6,DICPNT           ADD OFFSET
1293 08F6 6F06 00E2 R                 LD DICPNT,R6            UPDATE DICTIONARY POINTER
1294                                  NEXT
```

(Continued on next page)

```
1297                                *
1298                                        HEADER 5,LEA,ALLOT,     LEAVE
1302 0904 A9E1                      LEAVE   INC R14,#2              BUMP RETURN STACK
1303 0906 13EE                              PUSH @R14,@R14          SET INDEX = TERMINAL COUNT
1304                                        NEXT
1307                                *
1308                                        HEADER 1,IXX,LEAVE,     I
1312 0912 13FE                      I       PUSH @R15,@R14          PUSH LOOP INDEX ON DATA STACK
1313                                        NEXT
1316                                *
1317                                        HEADER 1,JXX,I,         J
1321 091E 53FE 0004                 J       PUSH @R15,4(R14)        PUSH LOOP INDEX OF NEXT OUTER LOOP
1322                                        NEXT
1325                                *
1326                                        HEADER 5,STA,J,         STATE
1330 092C 5F00 0460 R               ST8     CALL SYS
1331 0930 0005                              DW 5
1332                                *
1333                                        HEADER 4,HER,ST8,       HERE
1337 0938 53F0 00E2 R               HERE    PUSH @R15,DICPNT        PUSH DICTIONARY POINTER
1338                                        NEXT
1341                                *
1342                                        HEADER 7,EXE,HERE,      EXECUTE
1346 0946 97F2                      EXECUT  POP R2,@R15             GET COMPILARTION ADDRESS FROM STACK
1347 0948 1E28                              JP @R2                  GO DO IT
1348                                *
1349                                        HEADER 6,CRE,EXECUT,    CREATE
1353 0950 5F00 013E R               CREATE  CALL COLON
1354 0954 07C6                   R          DW ZERO
1355 0956 0996                   R          DW CNSTNT
1356 0958 0146                   R          DW SEMI
1357                                *
1358                                        HEADER 5,DOE,CREATE,    DOES>
1362 0960 5F00 013E R               DOES    CALL COLON
1363 0964 0652                   R          DW RFROM
1364 0966 03E6                   R          DW ENTRY
1365 0968 0434                   R          DW LITERL
1366 096A 000A                              DW 10
1367 096C 0578                   R          DW PLUS
1368 096E 0496                   R          DW STORE
1369 0970 0402                   R          DW SCODE
1370 0972 93E1                              PUSH @R14,R1            PUSH INSTRUCTION REG ON RETURN STACK
1371 0974 97F2                              POP R2,@R15             POP RETURN ADDRESS (OF CALL TO THIS ROUTINE)
1372 0976 9721                              POP R1,@R2              WORD THERE IS NEW INSTRUCTION ADDR
1373 0978 93F2                              PUSH @R15,R2            PUSH ADDRESS OF PARAMETER FIELD ON STACK
1374                                        NEXT
1377                                *
1378                                        HEADER 8,VAR,DOES,      VARIABLE
1382 0984 5F00 013E R               VARIBL  CALL COLON
1383 0988 0996                   R          DW CNSTNT
1384 098A 0402                   R          DW SCODE
1385                             R          NEXT
1388                                *
1389                                        HEADER 8,CON,VARIBL,    CONSTANT
1393 0996 5F00 013E R               CNSTNT  CALL COLON
1394 099A 03BE                   R          DW BUILD                MAKE A PRIMITIVE HEADER
1395 099C 08DA                   R          DW COMMA                ENCLOSE TOP OF STACK
1396 099E 0402                   R          DW SCODE
1397 09A0 97F2                              POP R2,@R15             GET ADDRESS OF THE CONSTANT
1398 09A2 13F2                              PUSH @R15,@R2           PUSH CONSTANT ON STACK
1399                                        NEXT
1402                                *
1403                                        HEADER 10,VOC,CNSTNT,   VOCABULARY
1407 09AE 5F00 013E R               VOCAB   CALL COLON
1408 09B2 0950                   R          DW CREATE
1409 09B4 03E6                   R          DW ENTRY
1410 09B6 08DA                   R          DW COMMA                MAKE HEADER AND ENCLOSE ADDRESS OF LAST ENTRY
1411 09B8 0960                   R          DW DOES
1412 09BA 0856                   R          DW CONTXT
1413 09BC 0496                   R          DW STORE                WHEN EXECUTED SET CONEXT VOCAB TO PARAM FIELD
1414 09BE 0146                   R          DW SEMI
1415                                *
1416                                        HEADER 1,:XX,VOCAB,     :
1420 09C6 5F00 013E R               COLON1  CALL COLON
1421 09CA 0862                   R          DW CRNT                 CURRENT
1422 09CC 0474                   R          DW FETCH
1423 09CE 0856                   R          DW CONTXT               CONTEXT
1424 09D0 0496                   R          DW STORE
1425 09D2 03BE                   R          DW BUILD                MAKE A PRIMITIVE HEADER
1426 09D4 02B2                   R          DW MOD
1427 09D6 0846                   R          DW C1SET
1428 09D8 0146                   R          DW SEMI
1429                                *
1430                                *
1431                                ********************
1432                                *
1433                                * COMPILER VOCABULARY
1434                                *
1435                                        HEADER 2,DOX,6,         DO
1439 09E0 5F00 013E R               DO      CALL COLON
1440 09E4 0434                   R          DW LITERL
1441 09E6 0458                   R          DW SYSDO                PLACE COMPILATION ADDRESS ON STACK
1442 09E8 03B4                   R          DW DOC                  COMPILE AND PUT DICTIONARY ADDRESS ON STACK
1443 09EA 0146                   R          DW SEMI
1444                                *
```

```
1445                                        HEADER 4,LOO,DO,        LOOP
1449 09F2 5F00 013E R          LOOP         CALL COLON
1450 09F6 0434              R               DW LITERL
1451 09F8 0442              R               DW SYSLOP               PLACE COMPILATION ADDRESS ON STACK
1452 09FA 03A4              R               DW ENDC                 ENCLOSE AND COMPUTE JUMP OFFSET
1453 09FC 0146              R               DW SEMI
1454                                    *
1455                                        HEADER 5,+LO,LOOP,      +LOOP
1459 0A04 5F00 013E R          PLOOP        CALL COLON
1460 0A08 0434              R               DW LITERL
1461 0A0A 043C              R               DW SYSPLP               PLACE COMPILATION ADDRESS ON STACK
1462 0A0C 03A4              R               DW ENDC                 ENCLOSE AND COMPUTE JUMP OFFSET
1463 0A0E 0146              R               DW SEMI
1464                                    *
1465                                        HEADER 5,BEG,PLOOP,     BEGIN
1469 0A16 5F00 013E R          BEGIN        CALL COLON
1470 0A1A 0938              R               DW HERE                 PLACE DICTIONARY POINTER ON STACK
1471 0A1C 0146              R               DW SEMI
1472                                    *
1473                                        HEADER 5,UNT,BEGIN,     UNTIL
1477 0A24 5F00 013E R          UNTIL        CALL COLON
1478 0A28 0434              R               DW LITERL
1479 0A2A 0420              R               DW SYSEND               PLACE COMPILATION ADDRESS ON STACK
1480 0A2C 03A4              R               DW ENDC                 ENCLOSE AND COMPUTE JUMP OFFSET
1481 0A2E 0146              R               DW SEMI
1482                                    *
1483                                        HEADER 5,WHI,UNTIL,     WHILE
1487 0A36 5F00 013E R          WHILE        CALL COLON
1488 0A3A 0A5A              R               DW IF
1489 0A3C 0146              R               DW SEMI
1490                                    *
1491                                        HEADER 6,REP,WHILE,     REPEAT
1495 0A44 5F00 013E R          REPEAT       CALL COLON
1496 0A48 0692              R               DW SWAP
1497 0A4A 0434              R               DW LITERL
1498 0A4C 042C              R               DW SYSWHL               PUT COMPILATION ADDRESS ON STACK
1499 0A4E 03A4              R               DW ENDC                 ENCLOSE AND COMPUTE JUMP OFFSET
1500 0A50 0A70              R               DW THEN
1501 0A52 0146              R               DW SEMI
1502                                    *
1503                                        HEADER 2,IFX,REPEAT,    IF
1507 0A5A 5F00 013E R          IF           CALL COLON
1508 0A5E 0434              R               DW LITERL
1509 0A60 040C              R               DW SYSIF                PLACE COMPILATION ADDRESS ON STACK
1510 0A62 03B4              R               DW DOC                  ENCLOSE AND PUT DICTIONARY POINTER ON STACK
1511 0A64 07C6              R               DW ZERO
1512 0A66 08DA              R               DW COMMA                SAVE A WORD IN DEFINITION FOR JUMP OFFSET
1513 0A68 0146              R               DW SEMI
1514                                    *
1515                                        HEADER 4,THE,IF,        THEN
1519 0A70 5F00 013E R          THEN         CALL COLON
1520 0A74 0938              R               DW HERE
1521 0A76 06A2              R               DW OVER
1522 0A78 0588              R               DW MINUS                COMPUTE JUMP OFFSET FOR "IF"
1523 0A7A 0692              R               DW SWAP
1524 0A7C 0496              R               DW STORE                STORE IN OFFSET WORD
1525 0A7E 0146              R               DW SEMI
1526                                    *
1527                                        HEADER 4,ELS,THEN,      ELSE
1531 0A86 5F00 013E R          ELSE         CALL COLON
1532 0A8A 0434              R               DW LITERL
1533 0A8C 0418              R               DW SYSELS               PUT COMPILATION ADDRESS ON STACK
1534 0A8E 03B4              R               DW DOC                  COMPILE AND PLACE DICTIONARY ADDRESS ON STACK
1535 0A90 07C6              R               DW ZERO
1536 0A92 08DA              R               DW COMMA                RESERVE A WORD FOR JUMP OFFSET
1537 0A94 0692              R               DW SWAP
1538 0A96 0A70              R               DW THEN
1539 0A98 0146              R               DW SEMI
1540                                    *
1541                                        HEADER 1,;XX,ELSE,      ;
1545 0AA0 5F00 013E R          SEMI1        CALL COLON              FIRST ENTRY OF COMPILER DIRECTIVES
1546 0AA4 0434              R               DW LITERL
1547 0AA6 0146              R               DW SEMI
1548 0AA8 08DA              R               DW COMMA                ENCLOSE ADDRESS OF SEMI IN THE DEFINITION
1549 0AAA 02B2              R               DW MOD
1550 0AAC 0838              R               DW C0SET                SET EXECUTE MODE
1551 0AAE 0146              R               DW SEMI
1552                                    *
1553                                    **********************
1554                                    *
1555                                    *
1556 0AB0 0002                 DP       DS 2                    NEW DICTIONARY ENTRIES ARE BUILT STARTING HERE
1557                                    *
1558                                    *
1559                                    *
1560                                        END
THERE WERE: 00000 ERRORS

COMMON LENGTH= 0000
```

End Listing

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

Just after watching Doc Dobb's nationally televised speech from the NCC in Texas, I was surprised to receive a collect call from the Old Man himself. "The lack of public domain software for 16-bit microcomputers is appalling!" remarked our Fearless Leader. "No one wants to shell out that kind of money just so they can stare at the operating system's sign-on message. I want you to institute a regular column in *DDJ* that will address the needs of 16-bit system users and promote the discussion and interchange of software."

Luckily, after laying down that sweeping and rather alarming mandate, the good Doctor had to hang up to go out on a house call (someone had choked on an Apple) and I was left to my own devices. After brooding on the subject for a while I have decided on the following objectives for this new column:

1) Implement, test, and publish a set of utility subroutines for the common 16-bit processors. Besides serving as building blocks for larger programs, these will demonstrate the relative merits of the different assembly languages and machines.

2) Translate the most useful programs from the CP/M-80 User's Group into 8086 code and place them in the public domain for further refinement by our readers.

3) Review commercial 16-bit software products, particularly applications and language systems produced by the smaller companies who can't afford to spend zillions of dollars on advertising.

All of the above is tentative, and I will be grateful for any suggestions and contributions.

Toolbox Installment #1

The accompanying listing contains two useful subroutines that we have to reinvent on every new microcomputer so that the program can interact with the operator. This version is in Standard Intel 8086 assembly language.

ASCBIN converts a number represented as an ASCII string (which presumably was read in from the console) into its binary equivalent. The field to be processed may contain the following characters: [0123456789+-.] . The routine is passed the beginning address of the ASCII field, and returns the two's complement value, the number of digits after the decimal point (if any), and the address of the first non-convertible character.

BINASC performs the opposite function, converting a signed binary number into an equivalent ASCII character string. The subroutine is passed the two's complement value and the address of a six-byte buffer where it can build the output, and returns the actual first byte address and length of the resulting string. ▶▶J

(Listing begins on next page)

ASCBIN and BINASC Listings

Conversion routines in 8086 assembly language.

```
                        title   '8086/8088 Utilities'
                        pagewidth 79
                        cseg
                ; Utility subroutines for 8086/8088
                ; Ray Duncan, Laboratory Microsystems
                ; July 1, 1982
                ;
                ;
                ; Convert ASCII to Binary
                ;
                ; Call with:    SI = address of ASCII string
                ;                    containing any of the
                ;                    characters: +-.0123456789
                ;
                ; Scans past any leading blanks.  Terminates
                ; on the first illegal character and returns
                ; its address.
                ;
                ; Returns:      AX = signed binary result
                ;               CX = number of digits after
                ;                    decimal, or -1 if no
                ;                    decimal point found.
                ;               SI = address of first
                ;                    non-convertible char.
                ;               Direction flag is cleared.
                ;               Other registers preserved.
                ;
0000 FC         ascbin: cld             ;clear direction flag
0001 52                 push    dx      ;save registers
0002 53                 push    bx
0003 33DB               xor     bx,bx   ;initialize answer
0005 B9FFFF             mov     cx,-1   ;and dec. place count
                ab1:                    ;scan past leading spaces
0008 803C20             cmp     byte ptr [si],' '
000B 7503       0010    jnz     ab2
000D 46                 inc     si
000E EBF8       0008    jmps    ab1
0010 56         ab2:    push    si      ;save addr of 1st char
                                        ;skip sign if any
0011 803C2B             cmp     byte ptr [si],'+'
0014 7405       001B    jz      ab25
0016 803C2D             cmp     byte ptr [si],'-'
0019 7501       001C    jnz     ab3
001B 46         ab25:   inc     si
001C AC         ab3:    lods    al      ;get next char to convert
001D 3C2E               cmp     al,'.'  ;check for decimal pt
001F 7504       0025    jnz     ab4
0021 33C9               xor     cx,cx   ;if decimal start to
0023 EBF7       001C    jmps    ab3     ;count trailing digits
0025 3C30       ab4:    cmp     al,'0'  ;make sure char is
0027 7216       003F    jb      ab5     ;in range 0-9
0029 3C39               cmp     al,'9'
002B 7712       003F    ja      ab5
002D 250F00             and     ax,0fh  ;mask off value
0030 93                 xchg    ax,bx   ;and add to previous
0031 BA0A00             mov     dx,10   ;data * 10
0034 F7E2               mul     dx
0036 03D8               add     bx,ax
0038 0BC9               or      cx,cx   ;count decimal places
003A 78E0       001C    js      ab3     ;if "." has been found
003C 41                 inc     cx
003D EBDD       001C    jmps    ab3
                ab5:                    ;end of numeric string
                                        ;found, point to first
003F 4E                 dec     si      ;non-convertible digit
0040 8BC3               mov     ax,bx   ;put answer into ax
0042 5B                 pop     bx      ;see if 1st char was sign
0043 803F2D             cmp     byte ptr [bx],'-'
0046 7502       004A    jnz     ab6     ;no, jump
0048 F7D8               neg     ax      ;yes, make answer negative
004A 5B         ab6:    pop     bx      ;restore registers
004B 5A                 pop     dx
004C C3                 ret             ;and return to caller
                ;
                ;
                ; Convert Binary to ASCII
                ;
                ; Call with:    AX = signed binary number
                ;               SI = address of 6 byte area
                ;                    to build ASCII string
                ;                    (sign and 5 digits)
                ;
                ; Returns:      SI = address of converted
                ;                    output string
                ;               CX = length of string
                ;               Other registers preserved.
                ;
004D 52         binasc: push    dx      ;save registers
004E 53                 push    bx
004F 50                 push    ax      ;save copy of value
0050 B90600             mov     cx,6
                ba1:                    ;fill buffer with spaces
0053 C60420             mov     byte ptr [si],' '
0056 46                 inc     si
0057 E2FA       0053    loop    ba1     ;loop until 6 stored
                                        ;now CX=0 and will be
                                        ;used for output count
0059 BB0A00             mov     bx,10   ;initialize divisor
005C 0BC0               or      ax,ax   ;if value negative
005E 7902       0062    jns     ba2     ;make it positive
0060 F7D8               neg     ax
0062 33D2       ba2:    xor     dx,dx   ;clear upper part of
                                        ;32 bit dividend
0064 F7F3               div     bx      ;divide by 10
0066 83C230             add     dx,'0'  ;make remainder into
                                        ;ASCII digit
0069 4E                 dec     si      ;back up thru buffer
006A 8814               mov     [si],dl ;store this digit
006C 41                 inc     cx      ;count char. converted
006D 0BC0               or      ax,ax   ;if all of value has
006F 75F1       0062    jnz     ba2     ;not been converted,
                                        ;loop again
0071 58                 pop     ax      ;get original value
0072 0BC0               or      ax,ax   ;was it negative?
0074 7905       007B    jns     ba3     ;no, jump
0076 4E                 dec     si      ;yes, store sign
0077 C6042D             mov     byte ptr [si],'-'
007A 41                 inc     cx      ;add to char. count
007B 5B         ba3:    pop     bx      ;restore registers
007C 5A                 pop     dx
007D C3                 ret             ;return to caller
                ;
END OF ASSEMBLY.  NUMBER OF ERRORS:  0.  USE FACTOR:  0%
```

BOOK REVIEWS

Starting Forth
by Leo Brodie
Prentice-Hall
366 pages, $15.95
Circle No. 302 on Reader Service Card
Reviewed by Richard L. Lozes

When I first picked it up, I thought that *Starting FORTH* was merely "cute." The author, it appeared, had gone overboard in illustrating with silly cartoons. Later, as I read about less familiar aspects of FORTH, I completely changed my mind; I found the cartoons an efficient and interest-boosting medium of communication.

Starting FORTH begins with two introductions, one for beginners and one for professionals. That duality persists, with special sections for computer novices and concise footnotes liberally sprinkled about, addressed to more or less advanced programmers.

Especially useful features of the book are the reviews of definitions appearing after each chapter and the last appendix, detailing all FORTH commands introduced. There is no index in the usual sense, but I did not miss it.

A word of warning for those more interested in polyFORTH – only the high-level, single-task commands of FORTH are presented here; the assembler, multiprommer, printer and disking utilities are excluded. But then, the book is *starting* FORTH.

Overall, I think Mr. Brodie has done his job well. Both the novice and the seasoned programmer will take interest and learn FORTH with the help of this book.

Discover Forth
by Thom Hogan
Osborne/McGraw-Hill
142 pages, $14.95
Circle No. 322 on Reader Service Card
Reviewed by George W. Jolly

Discover Forth is aptly named. I had been using Fortran and assembly language for ten years when my friends discovered Forth. They were virtually incoherent as they extolled the virtues of the new language. My reaction was, I think, typical. How could Forth possibly be worth the effort of learning to use Polish notation and such strange symbols? Eventually I came to share their enthusiasm, but it could have been much sooner if *Discover Forth* were available.

Author Hogan pleasantly explains the basics of Forth for the newcomer. Illustrations by Mary M. Milewski and section headings in a cheerful typeface make the book a pleasure to read. With some texts you work as hard reading the book as you do understanding the subject. *Discover Forth* is easy to read.

Fundamental to Forth is the concept of building-block programming. Each routine you write becomes a new feature to be used at will in your environment. Thus, you can tailor your programming tools to suit your own preferences. You can use the same routine as a command from the keyboard and as a subroutine in future programs without any special effort. You can even cause text on disk to be processed as if you were typing it, which opens many possibilities.

Once you learn the basic building blocks provided with your Forth system, you can evolve it into whatever system you want. However, the basic building blocks are somewhat unusual at first glance. Operations are performed in reverse-Polish notation on a data stack. Although many calculators operate this way, languages like BASIC and Fortran do not. There is no explicit assignment statement for variables; assignment is done by an operator. In fact, in the strictest sense, there are no statements in Forth at all, only definitions of words.

I found these differences between Forth and other languages only a minor stumbling block. Once you start using Forth, its advantages quickly outweigh the initial learning effort. *Discover Forth* would go a long way toward minimizing that initial effort.

Discover Forth describes the basic Forth language elements and how they are used. The stacks and the dictionary are clearly explained and illustrated, as are disk buffer I/O and structured branching. Most important, the philosophy of Forth programming is explained along with the basic ideas of top-down design. Forth-79 and fig-Forth are the basis for the discussion, with common differences of other systems pointed out for you. The required word set of Forth-79 is in an appendix for your reference. Advanced methods like DOES> are beyond the scope of this book.

While reading *Discover Forth*, you should be aware that advanced material is sprinkled into introductory dialogue. If you don't understand something, keep reading, you won't really need it yet. It is only there to whet your appetite for later. I like this because it helps keep the interest level up and gives the reader something to explore once the basics are over.

If you are curious about this strange language, and want an enjoyable look at it, try *Discover Forth*. Be careful, though; Forth is definitely habit-forming!

Programming Language Translation
by R. E. Berry
John Wiley & Sons (Ellis Horwood Ltd / Halsted Press)
175 pages, $41.95
Circle No. 312 on Reader Service Card
Reviewed by Edward T. Ordman
New England College

This is an unusual and excellent book that belongs on the shelf of anyone who wants to write a compiler, as well as any teacher of computer science courses beyond elementary programming. It is, however, sufficiently unconventional to require considerable sophistication from the reader. A first glance at the title and table of contents may suggest that this is a text for a standard compiler design course; it is not. Rather, it is a careful guided tour through a particular compiler.

Pages 115 through 152 (fine print – over 2000 lines) is the source code of a compiler and interpreter for Wirth's Pascal S. The rest of pages 85 to 171 are comments and explanations, including the pseudo-machine language for the stack-oriented machine and compiler output for sample programs (unsuccessful ones with diagnostics, and successful ones with pseudo-machine code output and run-time stack snapshots). Pages 9 through 84 contain a very abbreviated course in compiler and assembler construction, probably insufficient if one has not studied related material at greater length previously.

The extensive commenting and discussion of the compiler is very readable and will be helpful to anyone who has ever wondered how some particular feature of a compiler works. While Pascal S is not full Pascal (it lacks files other than INPUT and OUTPUT, GOTOs, sets,

and pointer variables), and the interpreter assumes that built-in functions (writeln, sin, cos, etc.) are provided elsewhere, it does have record structures, recursive procedures with value and variable parameters, and many other important Pascal features. Compiler directives are supported. (The source code, incidentally, is in a somewhat larger subset of Pascal – it uses sets heavily.) The author asserts that it is consistent with Data General NOVA (Lancaster P4) Pascal, as well as with UCSD Pascal for the Apple II and Western Digital Microengine.

One must jump around considerably, sometimes without warning, to read the book. The exercises to Chapter 1 are really questions about the source code 100 pages later. Page 106 explains the IF statement by showing what pseudo-machine code is produced; that code is not given in detail until page 169. There are a few errors: e.g., the explanation of procedure EMIT (insert in array CODE a pseudo-machine instruction with no operands) has been omitted on page 114. Despite these very minor shortcomings, the book will be a major help to anyone interested in how Pascal works and in how compilers work.

DDJ

SOFTWARE REVIEW

"The Working Forth Learning System"
Price: $57.95

Distributed on:
Two 8-inch, soft-sectored, single-density diskettes
System requirements:
8080 or Z80 microcomputer
Digital Research CP/M (tm) operating system
One flexible-disk drive
32 Kbytes RAM
Available from:
Laxen and Harris, Inc.
24301 Southland Drive, Suite 216
Hayward, California 94545

Circle No. 332 on Reader Service Card
Reviewed by Ray Duncan

Forth is a high-level language well-suited to many microcomputer applications because of its speed, portability and efficient use of memory. However, newcomers to it often become baffled and discouraged by the syntax (or lack of syntax), methods of disk access, and postfix notation, and they never reach a level of skill sufficient to exploit Forth's unique capabilities.

"Working Forth" is a computer-assisted tutorial that aims to boost the beginning Forth programmer past the initial awkward phase. The package was written by Kim Harris and Henry Laxen, well-known figures in Forth circles, whose firm provides consulting and educational services to many large corporations. Both are prolific authors and software designers (see September 1981 *DDJ* for examples), and have made major contributions to the development and standardization of the Forth language.

"Working Forth" is an interesting example of self-reference. This is a programmed instruction course about Forth which is written in Forth, and which is executed by a Forth interpreter/compiler that is in turn largely written in itself! Douglas Hofstadter fans will find plenty to ponder here.

Disk #1 of the set includes the following files:

- The executable Forth interpreter/compiler, which runs under the CP/M operating system.
- The source code for the nucleus of the interpreter/compiler. This is a standard ASCII text file that can be edited and assembled with the usual CP/M utilities.
- A text file containing the system documentation. This is a particularly nice touch – as you begin to modify and extend Forth for your own purposes, you can customize the documentation to suit yourself.

Disk #2 of the set includes many utilities and the programmed lessons. The former are for the convenience of the user in developing his own programs and include:

- miscellaneous debugging tools
- a line editor
- documentation aids
- partial or complete disk backup
- Forth 8080 assembler
- source code for additional data and control structures
- a decompiler

In my opinion, the utilities in themselves are worth more than the price of the package. They are friendly, well-documented and (so far as I have been able to discover) essentially bug-free. In addition, they serve as examples of excellent Forth coding style, showing how two veteran programmers have applied a wide range of techniques to create a variety of tools.

Getting started with the Forth programmed lessons is simple. Boot up CP/M, put disk #1 into drive A, and enter: FORTH <return>. The interpreter/compiler will load into memory, take control of the system, and print an identifying message. Replace disk #1 with disk #2 and enter: LEARNING LOAD <return>. From here on the system is self-instructing.

In general, the lessons organize information into nicely digestible chunks which will fit on one display screen. After a concept is presented, the learner is given access to the Forth interpreter so he can try out his new knowledge interactively. The tone of the lessons is chatty and affable, yet lucid.

The instruction begins with a brief review of the "philosophy of Forth," then a description of the virtual machine and how it is implemented. This leads into a discussion of Forth "words" or operators, how they are created, and how they are built into usable programs.

The user is led gently through concise but clear explanations of stacks, postfix or Reverse Polish Notation, comparison operators, constants, variables and control structures. By the time the user gets this far, he has built up a base of knowledge adequate to read most Forth programs intelligently and to write simple ones of his own.

The lessons subsequently become somewhat more technical, providing glimpses into the internal structure of the Forth runtime system. There is an introduction to "defining words" which allow the programmer to create new data types, as well as overviews of the representation of numbers, string handling, and output formatting. Next, the user is guided through Forth's mass storage access methods and how to use them properly. Finally, there are detailed descriptions and examples for each of the commands in the Forth line editor.

Once the user has absorbed all of the material in this learning system, he can go on to the excellent book *Starting Forth* by Leo Brodie which covers each topic in greater depth [*see Book Reviews, page 66*]. Laxen and Harris have taken pains to make their interpreter/compiler completely compatible with Brodie's programs and examples, so the reader can sit down with the book in front of a terminal and truly learn by doing.

In summary, the "Working Forth" lessons provide an inexpensive and enjoyable introduction to the Forth programming language. They will give you a clearer idea why people tend to polarize either violently against or fanatically in favor of Forth, and will let you judge whether the language is suited to your technical background, temperament, and applications.

As a bonus, the purchaser gets a highly functional Forth program development system complete with utilities, editor, and *all* source code. Laxen and Harris apparently have no desire to become language vendors, and state prominently in their documentation that "Working Forth" is meant to be used only as a teaching device. Such disclaimers aside, the fact remains that for a minimal cost you are actually getting a finely crafted tool, written by professionals, which can be used to create very sophisticated Forth applications. This is one of the great software bargains of the century.

DDJ

(See Figure 1 next page)

```
                    What's going on Here?

     Thank you for your interest in Learning Forth. This package is
designed to help you learn Forth quickly and easily. It uses the Forth
system described in the wonderful book STARTING FORTH by Leo Brodie. In
order to get the most out of this system, there are a few things you
should know.
     This software automatically decides when your screen is full and
stops displaying information. When this happens, the cursor will be at
the bottom of the screen just behind a > character. That is your signal
that the program thinks the screen is full. At this point you should
press the space bar and the information will continue.
     The number appearing on the upper right hand corner of your
display is called the Screen number. It is the place on the disk where
each particular piece of text resides. If you want to review any
particular screen and you know the screen number, you need simply type
the number followed by a space, followed by the word GO and a carriage
return. This will start the lesson at that particular screen.
     Often during the course of this session, you will be asked to try
something inForth, and we will exit from this package and leave you
in the Forth Interpreter. After you have completed the problem, or have
given up, you can return to the next instruction screen by typing G
followed by a carriage return. >
```

Figure 1.
Sample screen from the "WORKING FORTH Learning System" programmed lesson sequence.

(Continued from page 21)

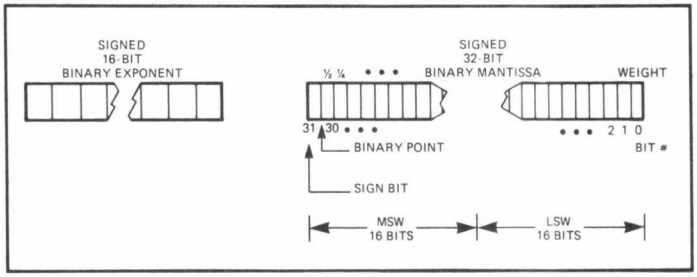


Figure 3.
Format of the FP#.

only about one and one-half weeks to write and debug the floating-point package described above. It has taken twice that time to write this article. The fact that Forth is structured and forces the user to build routines incrementally tends to foster error-free programming.

But best of all, Forth is extensible. It is application sensitive. This writer's ingrained prejudice for a good macro-assembler is fast disappearing.

DDJ

OF INTEREST

by Michael Wiesenberg

The trend of systems with two processors, eight- and sixteen-bit, continues with the **Dynabyte Monarch**. It has an 8-megahertz 8086 main CPU isolated on its own board, and a 6-MHz Z80B to handle input/output and other "housekeeping" tasks. Direct Memory Access (DMA) provides high-speed data transfers between disk, memory, and tape, completely independent of the two microprocessors. An optional 8087 math co-processor is available for heavy number crunching. With eight operating systems, Monarch runs, concurrently, virtually any 8- or 16-bit software. Available eight-bit operating systems are CP/M, MP/M-II, and OASIS; and 16-bit are CP/M-86, MP/M-86, OASIS-16, UNIX, and Business Basic (Basic Four). Two bus architectures are offered, either S-100 or Multibus. Up to 16 users are supported. The standard system includes 256 kilobytes of memory, a 19-megabyte 5¼-inch Winchester drive, an 8-inch 800K floppy disk drive, eight RS-232C asynchronous ports for modem or remote computer communications, *plus* one RS-232C asynchronous/synchronous port that has hand-shaking support for bisynchronous communications using IBM 2780/3780 protocols and 3270 terminal emulation, one RS-422 high-speed synchronous port, and one parallel (Centronics-compatible) port. (That's eleven I/O ports standard!) This system costs under $11,000, without optional additions like a 17-MB cartridge tape drive, memory expansion to 1MB, eight more serial ports, up to 76MB storage, and error detection and correction circuitry. Available languages include CBasic, Microsoft BASIC interpreter and compiler, Phase One Systems' BASIC interpreter/compiler, Ryan-McFarland COBOL, and, from Microsoft, COBOL-86, FORTRAN-86, and Pascal-86. **Reader Service No. 306.**

Similar features for considerably lower cost are offered by Zenith Data Systems' **Z100** desktop series. Processors are the 8088 16-bit and 8085 8-bit microprocessors. Two versions are available, the one-piece Z-120 with monochrome, green phosphor, non-glare screen, and the Z-110, a color computer without monitor. The "low-profile" model, with one internal 320K 5¼-inch diskette drive, monochrome graphics, and 128K RAM sells for $3249. All models have five S-100 slots, software-definable characters, blinking, underline, or reverse video cursor with relative and direct cursor addressing, a Z207 disk controller that simultaneously supports 5¼- and 8-inch drives, a 108-key keyboard with standard word processing keys, cursor control keys, a numeric keypad, and 13 function keys, two RS-232 ports in asynchronous mode, full or half duplex, and bisynchronous mode, one parallel port, and power switchable from 120 to 240 volts. Options include an extra internal 5¼-inch floppy disk drive, a 5¼-inch 5K Winchester drive, external 8-inch floppy disk drive (single- or double-density, single- or double-sided), memory expansion to 768K, and high-resolution color graphics. Available software includes Z-DOS (Zenith's implementation of Microsoft's MDOS), CP/M-85, Microsoft BASIC, Z-BASIC color extension, Multiplan electronic spreadsheet, and almost anything else that can be used by the IBM Personal Computer with PC-DOS or by any 8-bit CP/M. **Reader Service No. 316.**

Hewlett-Packard announced its lowest-priced personal computer, the **HP-86**, a modular unit with 128K of memory, expandable to 512K. It runs all HP Series 80 BASIC software, in addition to most CP/M software (with HP's plug-in CP/M system). The HP-86 is compatible with HP's personal computer modem, and a number of printers and graphics plotters. Using a Hewlett-Packard Interface Loop (HPIL), data gathered with an HP-41 handheld computer can be uploaded to any Series 80 computer. You can also get for the HP-86 a 9- or 12-inch display, a 270K flexible disk drive, an HP 82950 modem or HP Data Communications Pac, accounting software, dBASE II data base software, WordStar, SpellStar, MailMerge, VisiCalc PLUS, and graphics software. The HP-86 alone costs $1795, and a complete system costs under $3900. **Reader Service No. 326.**

Cyberdrive, from Cybernetics, is a 13.5- or 27-megabyte Winchester disk drive, with 6.4MB minicartridge tape for backup, for the IBM Personal Computer. It has an average seek time, they claim, of 33 milliseconds (including head settling), across more than five times as much data as other typical Winchester drives with seek times in the range of 100 to 200 ms. For $5750 (for the 13.5MB drive), you get all the hardware and software, and an extendable six-month warranty. **Reader Service No. 326.**

Interface Inc. offers 250K 5¼-inch floppy disk drives for the Apple II. The double-density, double-sided drive offers 40 tracks per diskette with half-tracking and 75ms average access times. You can get it (in any one of eight colors) for $375 for the single drive, and $850 for a dual-drive system with disk controller. **Reader Service No. 336.**

While we're talking storage for Apples, Innovative Data Technology offers the **SCDR-1050**, a tape drive that stores over 40 MB of data in IBM-compatible format, using nine-track 800- or 1600-cpi magnetic tape. The SCDR-1050 connects directly to an Apple II or III RS232C/CTTV-24 port with no modifications. The complete system includes an input/output controller with up to 32K of RAM, a TDF-4050 formatter, and IDT's TDI-1050 tape transport. It can transfer data for backup at about a megabyte per minute. One unit costs $8500, with additional daisy-chained drives at $4250 each. **Reader Service No. 346.**

You can get the new three-inch floppy disk drive and cartridge system, plug-compatible with standard 5¼-inch drives the **Micro-Floppydisk**, from Amdek. With up to one megabyte of storage on a dual-disk drive, it sells for $899. **Reader Service No. 356.**

Amdek also offers an X-Y four-color plotter, the **Model DXY**, for most computers using a parallel interface. It has a 10- by 14-inch plotting range, plots at 2.73 inches per second with a step size of .003 inch, comes complete with four pens, pen holders, and chart hold-downs, and costs $949. **Reader Service No. 366.**

Qunix from Quantum Software Systems is a Unix-like operating system for the IBM Personal Computer. Multitasking permits spooled output to a printer, and one to eight users are supported. Qunix is not the same as full Version 7 Unix, even though it uses message passing and generalized pipes and filters, but it is functionally equivalent in many ways, including the ability to run C programs that run under Unix. On the PC, any key can be redefined to represent a command sequence of variable length. A ROFF-like text-formatting program comes with Qunix. On-line help is available, and commands can be reassigned any names the user wishes. Qunix starts at $150 for a single-user system for the PC and 8088, 8086, and 6809 computers. The C compiler costs $350, including an 8086 assembler (which costs $100 by itself). A

full-screen editor (in addition to the supplied text formatter) costs $125, a BASIC compiler is $125, Pascal compiler $200, and FORTRAN compiler $125. **Reader Service No. 406.**

RMS (Record Management System) is a database management system (DBMS) from Washington Computer Services, for the IBM PC, or any computer using CP/M, CP/M-86, OS-9, Flex, or Uniflex. They say it is easily configured for any CRT, and consists of five machine language programs. Programming expertise is not required to use RMS, but its files can be accessed from most languages. A screen-oriented editor enters and retrieves records, with record layout determined by a user-definable data dictionary. A sort/merge program builds a sorted index, with direct access by key field, and multiple index files for ordered sequential access. Records can contain up to 50 fields, and be up to 1021 bytes long. A two-level record hierarchy lets any primary record "own" multiple secondary records. A file can be up to 64K in length. Field validation can automatically ensure entering of only valid data. A report program prints reports in sorted order, with certain records either explicitly skipped or included, output formatting specified, special forms accommodated, and headers and trailers included. A copy program merges RMS files, posts one to another, or changes the record format of a file. RMS prices range from $200 for Flex to $395 for IBM PC or CP/M. **Reader Service No. 416.**

Superfile from Elliam Associates, whose motto is, "It's what you can find that counts!", is another information retrieval system. Text files are created with any CP/M-compatible text editor or user program in "free form format." Superfile then searches these files, which can be on any number of different drives, using up to 64 key words, each of which can be up to 64 characters long. The search can be used with Boolean operators (AND, OR, NOT), and has a "close to" feature. That is, if a key word entry is not valid, Superfile displays a group of key words that are alphabetically close, so you can search semi-blindly for desired information. Superfile includes a customizing program, an alphanumeric sort, a merge program, and a program that splits records and files. Superfile also has a 150-page manual and a demonstration data base. It is supplied on various disk formats for Z80 systems with at least 48K and two floppy disk drives or a hard disk, and costs $196.50. **Reader Service No. 426.**

System 75 is a CP/M cross-assembler from Allen Ashley for the NEC 7500 microprocessor. The software system includes macroassembler, interactive editor/assembler, text editor, cross-reference generator, and facilities to offload the output. This development system is available on diskette for $150, including documentation, and user support by mail or phone. **Reader Service No. 446.**

Sinclair has announced new ZX81 software, on cassette and developed by ICL, a large British computer company, and Psion, a British software company. The software is available in the U.S. only by mail order, and ranges in price from $12.95 to $17.95. In the business applications category, **VU-CALC** is a Visi-Clone, and **VU-FILE** files and retrieves information. **Collectors' Pack** holds 400 records on hobby items (stamps, coins, etc.). **Club Record Controller** holds personal records of 100 people. The "Fun to Learn" series includes English Literature, Geography, History, Mathematics, Inventions, and Music, and the "Super Programs" series has games, quizzes, conversions, and houshold programs. Other games include Backgammon, six-level Chess, Flight Simulations, Space Raiders and Bombers, and Fantasy Games. There is also a Biorhythms. All cassettes, except the "Super Programs," require an add-on 16K RAM pack. **Reader Service No. 456.**

FORTH-79 from MicroMotion meets all provisions of the FORTH-79 standard, and is available for Apple, Z80 CP/M, and NorthStar DOS. It comes with a screen editor, macroassembler, string package, 32-bit integer arithmetic, and a tutorial and reference manual of some 200 pages, and costs $99.95. For floating-point and HIRES Graphics, add $40 for Apple and $49.95 for NorthStar (not available for CP/M). **Reader Service No. 476.**

GraFORTH, from Insoft, is a compiled graphics programming language for the Apple II. Three-dimensional color animation graphics employ rotation, scale, transposition, and perspective. Character set graphics come in full color, with variable character sizes and upper and lower case, and block print to manipulate large shapes. GraFORTH draws lines faster, says Insoft, than BASIC, and never breaks colored lines. It uses Turtlegraphics to draw lines. GraFORTH also has a software-based music synthesizer that adds music or sound effects to programs. GraFORTH comes with a 220-page tutorial manual, and costs $75. You'll need 48K, one disk drive with DOS 3.3, and Insoft recommends the 16K RAMcard. **Reader Service No. 486.**

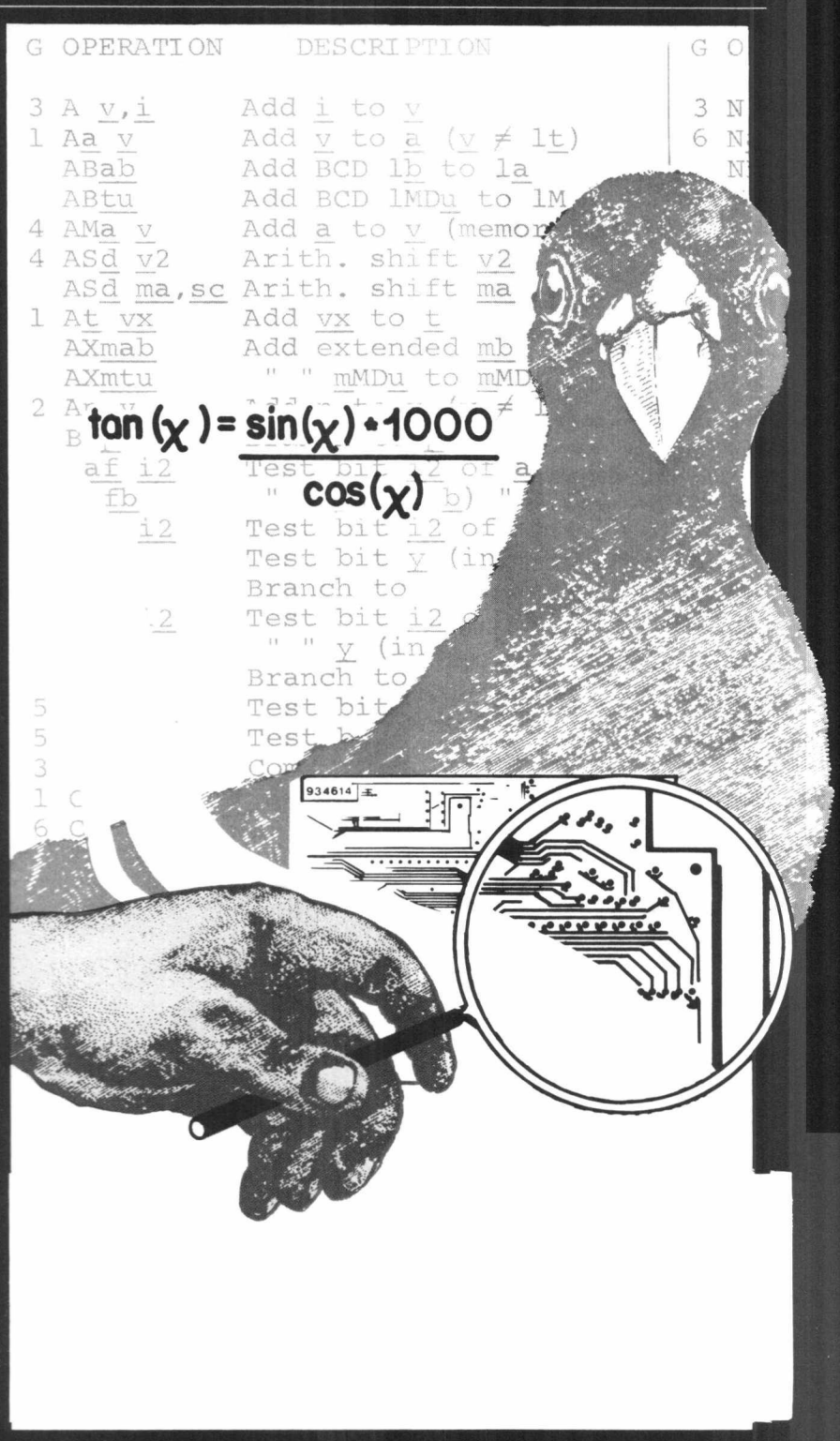
DATABS, from Softronics, was inspired by MIT's CLU language. It is a strongly-typed data abstraction language with UNIX-like command line arguments and I/O redirection. It comes on eight-inch CP/M disk, together with a 73-page manual, for $60. **Softronics APL** is a subset of APL, containing most of the functions and operators of the full language. Also for CP/M, it comes with a 112-page manual, and costs $350. Manuals alone are, respectively, $20 and $30. Write to Softronics, 366 Homestead Lane, Roosevelt, NJ 08555.

DDJ

72 October 1982 $2.50 ($3.00 in Canada)

Dr. Dobb's Journal

For Users of Small Computer Systems



Portable Pidgin for the Z80

68000 Cross Assembler

8086 Graphics Subroutines

and even more!

DR. DOBB'S JOURNAL
For Users of Small Computer Systems

October 1982 Volume 7, Issue 10

Editor – Marlin Ouverson
Assistant Editor – Fritzi Lareau
Contributing Editors –
Dave Caulkins, Dave Cortesi,
Ray Duncan, Gene Head,
Michael Wiesenberg
Marketing Director – Craig Harper
Marketing Assistant – Beatrice Blatteis
Advertising Director – Laird Foshay
Circulation Manager – Gloria Romanoff
Circulation Assistants –
Terri Marty, Linda Marohn
Art Director – Clifford West
Production Manager – Barbara Ruzgerian
Production Assistant – Jane A. McKean
Typesetter – Renny Wiggins
Cover Illustration – Al McCahon

© 1982 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Thomas H. Grohne, G. V. Elkins, Robert M. Connors, William J. McEown, Robert C. Luckey, Robert N. Stabler, DeWitt S. Brown, John B. Palmer, Burks A. Smith, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Howard E. Decker, Jim Wilson, Frank Lawyer, Chris Pettus, Paul Lutus, Ben Goldfarb, Rodney Black, Steven Weiss, Pat Phelan, John Brodie, James Fleming, Unison World, Thomas Davis, Jan Steinman, Ronald E. Johnson, G. Hunter, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor, Jr. **Sustaining Subscribers:** Pete Roberts, Steven Fisher Enterprizes. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Hofacker-Verlag, Tegernseer Strass 18, D-8150 Holzkirchen, West Germany. **Asia & Australia:** ASCII Publishing Co., Ltd, Aoyama Building 5F, 5-16-1 Minami Aoyama, Minato-ku, Tokyo 107, Japan.

CONTENTS

Articles

12 68000 Cross Assembler
by Allen Kossow
DDJ #68 described a homebrew, multi-68000 development system; readers responded with many requests for publication of the Fortran cross assembler mentioned in that article. The main program and subroutines will be presented in their entirety, beginning this month.

25 The Portable Pidgin
Z80 Macro-Assembly Implementation
by Herbert Gintis
Software portability often comes at the expense of size and speed. Author Gintis found Pidgin to be low-level, structured and portable. Here he presents his Z80 macro-assembly implementation intended to run under CP/M.

36 Simplified 68000 Mnemonics
by W. D. Maurer
Motorola's powerful and versatile 68000 microprocessor offers many features to the experienced programmer. But how does one ever become fluent in its complex assembler mnemonics? As this article points out, redesigning to a simpler mnemonic set may be a reasonable step to take.

Departments

6 Letters

6 Editorial

7 Dr. Dobb's Clinic
Debuggers, Nested SUBMITs and PC BASIC

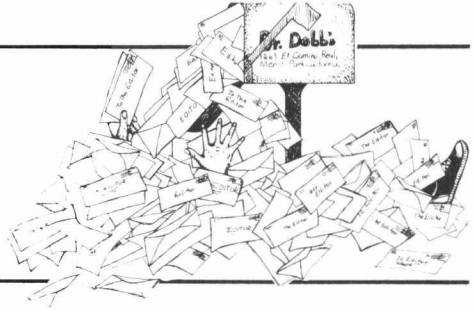
42 CP/M Exchange
RCP/Ms, MBOOT3 and Downloading MODEM

51 16-Bit Software Toolbox
8086/88 Trig Lookup Functions

55 Of Interest

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, Box E, 1263 El Camino Real, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025 • 415/323-3111

LETTERS



Dear Gary Kildall . . .

Dear *Dr. Dobb's*,

I found the article comparing CP/M-86 with MS-DOS (and CP/M-80) quite interesting, though I certainly agree with the conclusion that it is disappointing that Digital Research and Microsoft can't come up with anything better than a copy of CP/M-80. In my opinion, the biggest item missing from all three of these systems is I/O redirection.

Recently, Digital Research ran ads in a newspaper in this area for people to work on CP/M for the Motorola 68000. Are we in for yet another copy of CP/M-80? That would be rather like running a Ferrari engine on one cylinder. (It is gratifying, however, that Digital Research has finally recognized that someone besides Intel makes microprocessors!) I would like to suggest to Digital Research that they use Microware's OS-9^{tm} operating system (for the 6809) as a model for CP/M-68K (or whatever they call it), rather than giving us yet another copy of CP/M-80.

Sincerely,
Jim Howell
5472 Playa Del Rey
San Jose, CA 95123

Micro Compiler in Brobdingnag

Dear Marlin Ouverson,

Enclosed is an article and program submitted for your review and inclusion in *Dr. Dobb's*, if you so desire. Let me add that I enjoy your magazine very much, have all the issues from #1, and it would be the last magazine I would drop. (I get over thirty engineering or computing magazines.)

I have modified the Small-C compiler to work on a 370 under VM. How's that for reverse engineering? It also has FOR, GOTO, two-dimension arrays, etc.

Regards,
Chris L. Torkildson
13791 Heywood Ct.
Apple Valley, MN 55124

Dear Mr. Ouverson,

I have been using Small-C since the Software Works first offered it for CP/M. I recently got enough memory to allow it to compile itself correctly, and have started implementing all the changes suggested

(Continued on page 59)

Editorial

Going Public . . .

Dr. Dobb's Journal has always championed the cause of public-domain software. It was accidentally founded by a group of idealistic hackers who only wanted to write and publish a tiny Basic interpreter. Their challenge was to fit it in about 4K; their commitment was to put it in the public domain.

This idea – doing good work and letting the public have it with no strings – was so good that readers who managed to get their hands on a copy just wouldn't let *DDJ* finish its intended three-issue life cycle. Instead, seven years later it is the landmark publication for systems software.

One reason for practically giving away perfectly good programs is to enable individuals to build a library of tools for their own use. In the larger sense, of course, commercial products would be greatly enhanced by drawing on some of the truly superb items that have been launched into the public domain. In fact, a number of good packages for sale these days may find their roots in a *Dr. Dobb's Journal* listing, or some other non-proprietary source.

This altruistic influence on end-user technology has, we think, improved the state of the art. The *DDJ* community waves the public-domain flag because we think it is important. With this kind of material, the available computing power is greater than would otherwise be true.

Many generous and sympathetic authors should be thanked for making *DDJ*'s mission successful. In addition to our present contributors, we remember the work of Dennis Allison, Steve Wozniak, Gary Kildall, Tom Pittman, Ron Cain, Li-Chen Wang and Ward Christensen. They are among the many fine people who have used *DDJ* to make important contributions to the microcomputing community.

Unfortunately, the creation of public-domain software is not as popular a pastime as it once was. Maybe, in the beginning, we had more of a sense of community – wonderful machines with unlimited potential, but with nothing more helpful than a row of switches on the front panel. And if you think today's documentation is bad . . . it took a major team effort just to get the things operating.

Now, software protection schemes seem to get more creative thought than the programs they were designed to protect. Instead of sharing resources, one sees even (or perhaps especially) inexperienced programmers becoming canny to any possible profit to be gained or lost. Fortunately, there is still a body of people willing to share their work for just the good feeling, or the prestige, or the common cause of more public-domain software.

In the interest of building the public domain, we encourage authors not to reserve rights to programs published in *DDJ*, even though we will publish ones which reserve commercial rights. If the technical growth potential of computers is going to be realized, it won't be by those large companies with heavy investments in current technology – it will be by individuals working together to push beyond the limits of today's product line. When you hear one of us refer to the *DDJ community*, that's what we mean.

– Marlin Ouverson

DR. DOBB'S CLINIC

by Dave Cortesi

The Very Last SUBMIT Item

For months we have been droning on about the SUBMIT command of CP/M, telling you how useful it can be and showing how to increase its usefulness with fixes and small utilities. In June we showed the utilities PAUSE and BEEP, which increase the operator's control over a submitted job. In August we brought you Robert Pasky's nicely-integrated set of patches that make SUBMIT handle lowercase input, control characters and null lines correctly. With them applied, you can submit command streams to ED and PIP for unattended execution. In September we presented Don Wright's QUITIF program, which gives the designer of a SUBMIT file the ability to check for some errors.

Now, courtesy of Digital Research, we SUBMIT users have the ultimate in SUBMIT patches. The July/August issue of *Microsystems* reproduces, unedited and without commentary, a set of applications notes, the work of the Technical Support Group at Digital Research, Inc. One of these is a lengthy patch which causes SUBMIT to append its output to an existing $$$.SUB file, instead of replacing that file. The result is that we can include a SUBMIT command in a submitted job. The inner submit file will be appended to the active submit file; when it completes, the outer submit file will be resumed with the line following the SUBMIT command.

Nested submits! And they come so easily, too. When SUBMIT runs, it places the submitted commands, one per 128-byte record, in reverse order in the file $$$.SUB. This was originally done, we think, for the convenience of the Console Command Processor (CCP). The CCP consumes records from $$$.SUB from the end of the file toward the beginning. That lets it use the "record count" byte of the directory entry for $$$.SUB as its pointer to the next record of the file. It couldn't keep such a pointer in storage, because there is no location in storage that is proof against overlay by an application program.

But this use of $$$.SUB makes the file work like a push-down stack. The CCP pops records off the end of the file. The new patch makes SUBMIT push records onto the end of the file, rather than erasing the file and building a whole new one. Nesting of submitted jobs follows automatically.

The Digital Research patch, with our own comments, is shown in Listing 1 (page 58). Assemble it, then use the I and R commands of DDT to overlay its hex file onto a copy of SUBMIT.COM (one with the Pasky Patches applied to it). The updated SUBMIT command does just about everything we could ask of it, so with this item we are turning the whole matter of CP/M SUBMIT over to Gene Head and the *CP/M Exchange*.

Debugging the Debuggers

We've had some letters from people who have problems with the Digital Research debug utilities, DDT and ZSID. L. Barker sent us a note on two holes in ZSID. It doesn't interpret the LDIR instruction correctly when the operands (the two strings defined by the HL, DE and BC registers) overlap on each other. The LDIR (long move) instruction of the Z80 copies a source string into a target string. When the target string overlaps on the source string (as when the source begins at 0200h, the target at 0201h, and the length is 255 bytes), LDIR becomes a Replicate instruction, duplicating the leading byte of the source into the target over and over. ZSID doesn't handle it right, though; it stops after copying one byte.

Barker also noted that the disassembler in ZSID can't seem to handle the "LD A,R" and "LD R,A" instructions. Its List command won't display them and its Assemble command won't accept them.

Gavin Brickell of Auckland, New Zealand, thought that he'd found a bug in DDT, but he hadn't. It appeared that DDT was modifying a word in storage without any reason. However, it had a reason: the word in question was the top word of the stack as defined by the program under test. DDT uses one word on the program's stack in at least three points during the execution of the Go command. The remarkable thing is that DDT manages to get by with only one word of the program's stack; it seems not to need any more.

Dana Trout of Goleta, CA, wrote in with a method for changing the number of the RST vector that DDT uses for breakpoints. One of the Digital Research applications notes mentioned above covers the same area. We think that DDT's use of RST 7 should be left alone if at all possible. DDT (and SID, and ZSID, and any other debugging tool) is going to use *some* restart vector, and RST 7 is the one that is documented as being for this use. The great majority of CP/M systems will leave RST 7 free for debuggers. A modified debugger, exported to another system, is quite likely to crash that system by conflicting with a legitimate interrupt vector. Furthermore, there's a good reason for using RST 7 for debug breakpoints: most hardware will return FFh when asked to read from RAM that doesn't exist. FFh is the opcode for an RST 7 instruction; if that is the breakpoint vector, a wild branch to nonexistent memory will cause an automatic breakpoint!

Finally, Nick Hammond of Pebble Beach, CA, contributed a cute trick that can be played with DDT. DDT could be useful for poking around in the innards of CP/M, if only it would leave things alone. "The problem with DDT," Hammond writes, "is that it changes things when it gets control. Location 5 no longer points to the BDOS except via a circuitous series of jumps, and the CCP is missing, having been overwritten." Hammond supplies a technique for getting a copy of DDT into the transient area below the CCP, and getting control into it with the BDOS vector and the CCP intact. It works like this.

First, invoke DDT as a command, with its own .COM file as argument:

```
A>ddt ddt.com
```

DDT will start up and load DDT.COM. Give the Go command with no address; control will enter the loaded copy. It will relocate itself just below the first-started copy of DDT, clear of the CCP, put its own vector into the RST 7 location, and prompt for a command. Use the Assemble or Substitute command to put an RST 7 instruction at location 0100h. Then use the G0 or a control-c to do a warm start. The CCP will be reloaded over the first copy of DDT, but the second copy will remain. Then create and call a null command file as a means of causing a branch to location 100h, where you stored a RST 7:

```
A>save 0 null.com
A>null
```

Control will pass to the RST 7 at location 100, which will return control to the second copy of DDT. You can now use it to peer about the system, observing low storage and the CCP in their native states.

Ask Uncle

"I have a Diablo 1620," says Ernest

Knipp of Houston, TX, "and the Courier-10 printwheel has a pound-sterling symbol. I can't get this symbol to print. Is there any way to reach this unreachable character?"

You betcha, Ernest. There are 96 spokes on the Diablo printwheel. Ninety-four of them print in response to the 94 printing chracters of the ASCII code. Two of the spokes correspond to the ASCII space (32h) and the DEL (7Fh) codes. Those two can't be reached by sending the printer a single ASCII code. When it gets a space, the printer just advances the carriage; it doesn't use the printwheel at all. And it conforms to the ASCII standard by ignoring DEL characters entirely.

However, the manual for our 1650 printer says that the sequence ESCape, "Y" will print the spoke matching 32h, and ESCape, "Z" will print the one matching 7Fh. In other words, if your printer receives the *two* bytes 1Bh, 59h, it should print the sterling symbol. If it receives the bytes 1Bh, 5Ah, it should print a logical-not symbol.

Assorted Grumps

A person who would rather remain anonymous has told us that he had it from an equally anonymous, but supposedly well-informed, source that the code of the IBM PC's BASIC interpreter was indeed produced by an 8080-to-8086 translator program. That's all very well, but we'd rather see some evidence. Surely somebody out there has a PC, a suspicious mind, experience with the typical output of translator programs, and a few hours to spare. The MSDOS command DEBUG will disassemble storage. Use Ctrl-PrtSc to start a log of its output on the printer.

Actually, there are very good reasons why Microsoft would translate an existing interpeter rather than write a new one. Their MBASIC is a mature, well-debugged piece of code. Translation would let them produce a reliable product quickly and cheaply. Reliability, quick development, and low cost are three qualities that are conspicuously absent when you build a large piece of software from scratch. One can see how the "technical" consideration of execution speed could easily take a back seat to these desirable "management" criteria. . . .

Burke Smith of Shawnee Mission, KS, has found a bug in that mature product, MBASIC. It only seems to surface in version 5.1 of BASIC-80, and in IBM version D1.00. Try this sequence:

```
LET A#=0
LET A#=-A#
PRINT INT(A#),FIX(A#)
```

Smith claims that in the noted versions (but not in the more common versions 4.51 and 5.2) two answers of -1 will be diplayed, "rather than the zeroes that any reasonable person would expect."

Smith goes on to blame the problem on Microsoft's choice of binary floating-point rather than decimal float using a BCD representation. We think that's a red herring. We've seen a lot of misguided complaints along this line; the writers seem to think that there is something peculiar about the binary number system that makes it more prone to truncation errors ("round-off errors") than decimal is. Tain't so; a digit is a digit, no matter what your radix. The problem is the limited precision of the representation, not the number base. A decimal representation of a real number is just as prone to truncation errors as a binary, octal or hexadecimal one is.

There are standard techniques that will mitigate the effect of truncation errors. Hardware designers can incorporate an extra digit of precision in the temporary results held in machine registers, and round any bits that turn up in such a "guard digit" back into the result before storing the result. Few people remember now, but the initial design for the IBM 360 line lacked guard digits in the floating-point ALU; the first machines had them added in the field as engineering changes. Good scientific calculators get the same effect by carrying several low-order digits that don't show in the display. MBASIC could have

SUBMIT.COM

```
; A patch to SUBMIT.COM which forces it to open an existing
; $$$.SUB file for appending, rather than erasing it.  The
; CCP reads that file in reverse order.  Thus a SUBMIT within
; a submitted file will "push" new records onto the end of the
; file, from whence the CCP will "pop" them.  The result is
; correct execution of nested SUBMIT commands.
;
subfcb  equ     05BBh   ; FCB for $$$.SUB
BDOS    equ     0005h
OPEN    equ     0211H   ; open-file subroutine of SUBMIT
;
        org     022Dh   ; the erase-file routine of SUBMIT
;
; It would seem that this location is called when an attempted
; open of $$$.SUB succeeds, showing that the file exists.
; Previously, this code would have erased the file.
;
opsl    lda     subfcb+15 ; open ok if extent not full
        ral               ; extent full if record count=>80h
        rnc               ; exit if count < 80h
        lxi     h,subfcb+12 ; extent number in FCB
        inr     m         ; try the next extent
; Code to invoke or re-invoke the create-next-extent routine
; of SUBMIT--which will call "opsl" above if the extent exists?
ops     lxi     d,subfcb ; open first (next?) extent
        jmp     create
;
; Routine to create the $$$.SUB file
;
        org     025Dh
create  call    OPEN    ; test $$$.SUB by opening it?
        inr     a       ; FFh -> 00h if open fails
        jnz     opsl    ; do above code if file exists
        lxi     d,subfcb ; file does not exist, make it
        mvi     c,22
        call    BDOS
        adi     01h     ; set carry flag if retcode=FFh
        ret
;
; Replacement code for original logic to open $$$.SUB
;
        org     04FEh
        call    ops     ; open last extent of $$$.SUB
        jc      0517h   ; error if open failed
        lda     subfcb+15 ; use the FCB "record count"
        sta     subfcb+32 ; ..to set the "current record"
        jmp     051Dh   ; continue with original code?
;
        end
```

achieved the same effect in software; we don't know if it tries to.

A decimal representation is better in just one way: it will get truncation errors on the numbers that people expect it to. Nobody is too surprised when the expression ((1/3)*3) does not yield a result of 1.0; most people can picture the string 0.333333 . . . being truncated and multiplied by 3 to yield 0.999999. The trouble with binary (or any other radix) is that some fractions that have finite representations in decimal become repeating "decimals" in another radix (and vice versa). When the system converts such a constant to its internal radix, it loses some precision before any calculations are done at all. For commercial arithmetic, the only answer is to use a fixed-point decimal representation like the ones available in PL/I and COBOL compilers. Floating-point arithmetic should *never* be used for commercial calculations, no matter what radix it is based on.

John Palmer of Boonville, CA, is grumpy about his new CP/M-86 system, which he procured from Godbout along with the Godbout 8085/8088 dual processor board. "The Godbout hardware is very good," he writes. "If you buy all the stuff from Godbout it will run." However, problems arose when he had to modify the Godbout BIOS. The first problem was that "the BIOS is very hard to modify without the Sorcim anti-Intel ACT assembler." That would make us grumpy, too. There is no excuse for distributing a BIOS that requires an assembler that isn't part of the distributed system. "The source code for the BIOS is on the diskette," Palmer reports, "but you cannot use it until you buy Sorcim's ACT-86 for $175." Palmer thinks this is a tacky way of selling assemblers, and we agree.

Passing the Hat

We're scraping the bottom of the mailbag again, readers. You've all been very flattering in your praise of this column, but you aren't *contributing*. We need questions, like Knipp's. Pitfalls and Warnings, like Palmer's. Bugs, like Barker's. Patches, like Pasky's. Discoveries, like Hammond's. Puzzles. Confusions. Grumps. Cheap software. *Material!* Please?

DDJ

68000 Cross Assembler

by Al Kossow

Allen Kossow, Medical College of Wisconsin, Department of Physiology, 8701 Watertown Plank Road, Milwaukee, WI 53226.

The June 1982 issue of *DDJ* contained a brief description of the computer system which is being developed by myself, Darryl Uchitil, and Jim Hannas as a test bed for system software and hardware using the Motorola 68000 microprocessor. The reader response to both *DDJ* and myself as a result of that article has been interesting and informative to Jim, Darryl and myself, and I would like to thank all of you who have called or written me since June. I will present at the end of this article a summary of answers to some of the most frequently asked questions about our system for the benefit of those who have not tried to contact me or who have written but have not yet received my reply.

Cross-Assembler Description

This article will describe the cross assembler that was mentioned in the June 1982 issue of *DDJ*. It was developed because we needed a way to write small programs for execution on a single 68000 processor to test the hardware and to test the interface between the 68000 processors and the I/O bus control processor. The cross assembler for the 68000 has been run on two different PDP-11 systems, one running RT-11, and one running RSX-11M. It has also been run on a VAX/VMS system under the applications migration executive. It should be possible to adapt the assembler to other machines by recoding assembly-language subroutines for 32-bit arithmetic and modifying the routines that reference octal constants, but as of this writing, no attempt has been made to do so.

The cross assembler consists of a main program and a collection of subroutines, each performing part of the assembly process. The cross assembler is a two-pass assembler; the source code is read twice by the program. The first pass through the source code picks up all of the labels, and the second pass generates the listing and object files. All operations to files are handled by individual subroutines. These subroutines are SOURCE, LIST, and OBJECT. All machine-dependent routines for opening and closing of files are located in these routines.

Each of the three routines for the source, listing and object files is called when the program is first started, to determine where each is to be sent. Some checking is done to determine if just a carriage-return was typed, so that there is a default of only a listing to the terminal and no object file is established. Once the source for the assembler and the destinations of the listing and object are established, the assembler starts the assembly by setting the pass number to one and calls the subroutine PARSE.

PARSE scans the input line and determines if the statement is just a comment. If it is not a comment, it splits apart the label, opcode and operand fields of the statement, and returns pointers to these fields to be used by the statement evaluation routines.

After the line is parsed, the statement opcode is evaluated by the subroutine PRCESS. PRCESS calls DECOPC, which takes the characters pointed to by the opcode pointer from PARSE and attempts to find an opcode that matches. If an opcode is found, the values of one or more skeletal opcodes (opcodes without effective addresses or sizes) are returned.

After evaluating the opcode, the operands are evaluated to determine their general type. All "simple" operands, such as registers, are evaluated immediately, while "complex" operands (i.e., operands containing labels) are not evaluated.

If a valid opcode skeleton was returned, there will be a number returned by DECOPC which represents a general way in which to evaluate this type of operation. This number is used by PRCESS in a multi-way branch to different sections of code for evaluation of the opcode. Once in the specific section of code for the opcode, the operands are checked for validity in the opcode, and if the operand was "complex" it is evaluated and a value is returned.

Each specific section of code for an opcode builds up the opcode skeleton, filling in the size and effective address fields as necessary for that type of instruction. The result of this evaluation is an array of 16-bit values which represent the result of that line of the assembly, and a count of the number of words generated by that line.

At the end of the opcode processing routine is the section of code that handles labels. If a label was detected and is valid for the opcode type, the current value of the location counter is placed in the symbol table entry for that label.

If this is the second pass, the object and listing is generated based on the values in the instruction word array; otherwise, the next line is fetched and the evaluation process is repeated. After the last line of the program is read, the input file is closed, and a routine is called to print the contents of the symbol table.

Frequently Asked Questions About The Multi-68000 System

Q: Can I get a copy of your assembler?

A: If you don't feel like typing in the source code in this article, I can send you the source as it appears here on an 8" single-sided, single-density floppy disk in either CP/M or RT-11 format for $25 on an as-is, no-support basis. I can provide the source on a CP/M disk, but I have no way to compile it or test it there.

Q: What is CLICS? Where can I get it?

A: CLICS is a collection of subroutines for 2-D graphics developed by Mike Garrett while he was with the Department of Defense. Since the time I wrote the article, I have heard that Mike left DOD and started a company selling graphics software written in C. I have not been able to reach him to determine if this is true or not. The CLICS software I have been working with came off of the 1980 Spring DECUS RSX-11 special interest group symposium tape. This tape is available from the DECUS library.

Q: When/where can I buy one of your systems?

A: The notion of selling this system has been discussed a number of times between the three of us, but we just don't want to get into the computer business. The schematics and blank PC boards will probably be available from us on an as-is, no-support basis, along with the software we develop, but we have no plans to sell or support systems.

Q: Why didn't you build your system on a "XYZ" bus?

A: I had expected much more flack about using the PUNIBUS than I actually received. The primary reason for not using a commercial bus was the desire to build a system which had a few large (8" by 15") cards to minimize the number of cards to build. We also probably would not have used a special bus if we had planned on selling the system once we finished it.

In closing, I would like to thank all of you that have taken the time to call or write me about our project, and will promise to keep you informed of our progress in future issues of *DDJ*.

DDJ

(Listing begins on page 14)

68000 Cross Assembler (XASM)

(Text begins on page 12)

Due to its length, this listing will be continued next month.

Listing One

```
      PROGRAM MAIN
C
C     M68000 CROSS-ASSEMBLER MAIN PROGRAM
C
C
C DATE:
C     FEBRUARY 10, 1981
C
C REVISION:
C     X1.0    (EXPERIMENTAL PRE-RELEASE)
C

C
C AUTHOR:
C             Allen Kossow
C             Medical College of Wisconsin
C             Department of Physiology
C             8701 Watertown Plank Rd.
C             Milwaukee, WI 53226
C
C PGM NOTES:
C
C             Symbols are unique to 8 chrs.
C             Symbol table size is currently set to 512.
C             Symbol table size is determined by the memory constraints
C             of the machine it's running on. The current symbol table
C             size works under RT-11 with 64kb of memory and under RSX-11.
C
C             This program follows most of the conventions of the
C             Motorola 68000 assembler with the following exceptions:
C
C             1) no macros or pc relative addressing (except branches)
C             2) generalized (i.e., simple) object file format
C             3) listing format similar to that of MACRO-11
C             4) doesn't handle adr regs as special cases (i.e., MOVEA is just a MOVE).
C
C


C.....  LOGICAL UNIT DEFINITION
C     1 = SOURCE FILE
C     2 = OBJECT FILE
C     3 = LIST   FILE
C     5 = KEYBD
C
C
      IMPLICIT INTEGER (A-Z)
      BYTE NAME(8),SYMFLG(513)
```

(Continued on column 2)

```
C
      COMMON /LST/ LUNIT,PASS,NAME,NOPAGE,NOLINE,MEFLG,IERCNT
C
      COMMON /SYMT/STIND,SYMADR,PC,NOSYM,NEWPC,SYMFLG
C
      COMMON /PRSE/ OPPTR,MODPTR,OPNPTR,LABEL,CMTPTR
     +,PRFLG,SCANPT,OPCLEN,OPNPT2,IMODE
C
      COMMON /SRC/ LNELEN,ISERR,NOCARD,SRCLNE
C
      COMMON /OBJOUT/ OBJBUF,OBJWC,LFLG,RFLG,DBFLG
C
      COMMON /OPCPTS/ OPTYP,OPSKEL,OPSK2,OPIDX
C
      COMMON /CNVT  / WORD,PL
C
      COMMON /HEXFLG/ ENDFLG,HEXWC,HEXPC,OLDPC
C
      DIMENSION OBJBUF(40)
C
      INTEGER*4 PC,NEWPC,SYMADR(512),HEXPC,OLDPC
C
      BYTE ERR,SRCLNE(81),LABEL(8),PL(132)
C
C.....  TELL FORTRAN TO IGNORE INTEGER OVERFLOWS ON
C     MULTIPLY AND DIVIDE.
C

C
      ERR=128
C
C.....  THE FOLLOWING CALL IS NO OP'ED OUT FOR F4P
C
C     CALL SETERR(1,ERR)
C
C.....  INITIALIZE VARIABLES
C
5     NOPAGE=0
      RFLG=1
      LFLG=1
C
C.....  OPEN FILES
C
      CALL SOURCE(1)
      CALL LIST(1)
      CALL OBJECT(1)
C
C.....  DO PASS 1
C
1     NOSYM=0
      PASS=1
      CALL I4CLR(PC)
      DO 10 I=1,8
      NAME(I)=32
10    CONTINUE
```

(Continued on column 3)

```
C
C.....  READ ONE LINE OF SOURCE FILE
C
15    CALL I4CLR(NEWPC)
      CALL SOURCE(2)
C
C.....  IF EOF DETECTED DO PASS 2
C
      IF(ISERR.EQ.1) GOTO 20


C
C.....  RESET MULTIPLE ERROR FLG
C
      MEFLG = 0
C
C.....  PARSE SOURCE LINE
C
      CALL PARSE
C
C.....  IF NULL LINE GET NEXT LINE
C
      IF(PRFLG.EQ.0) GOTO 15
C
C.....  PROCESS SOURCE LINE
C
      CALL PRCESS
C



C.....  IF END DETECTED DO PASS 2 ELSE GET NEXT LINE
C
      IF(ISERR.EQ.1) GOTO 20
      I=JADD(PC,NEWPC,PC)
      GOTO 15
C
C.....  DO PASS 2
C
C
C.....  REW SOURCE SET TO PASS 2 AND RESET PC
C
20    CALL SOURCE(3)
      PASS=2
      IERCNT = 0
      CALL I4CLR(PC)
      CALL I4CLR(HEXPC)
      CALL I4CLR(OLDPC)
C
C.....  FLUSH PRINT BUFFER IN CASE ANYTHING LEFT
C.....  FROM LAST ASSEMBLY
C
      DO 25 I=1,132
25    PL(I) = '40
C
C.....  INITIALIZE OBJECT BUFFER
C
```

(Continued on page 16, column 1)

68000 Cross Assembler (XASM)

(Listing continued, text begins on page 12)

Listing One

```
      ENDFLG = 0
      HEXWC  = 0
C
C....  PRINT FIRST PAGE HEADING
C
      CALL NEWPAG
30    CALL I4CLR(NEWPC)
      OBJWC = 0
      CALL SOURCE(2)
C
C....  EOF DETECTED
C
      IF(ISERR.EQ.1) GOTO 50
C
C....  RESET MULTIPLE ERROR FLG
C
      MEFLG = 0
C
C....  PARSE LINE
C
      CALL PARSE
C
C....  PRINT A LINE OF ONLY COMMENTS NORMALLY
C
      IF(CMTPTR.EQ.1) GOTO 40
C
C....  CHECK FOR PARSING ERRORS
C
      IF(PRFLG.EQ.0) GOTO 30
C
C....  PROCESS IT
C
38    CALL PRCESS
C
C....  GENERATE LISTING
C
40    CALL LSTLNE
C
C....  CHECK IF THERE IS OBJ CODE TO GENERATE
C
      IF(OBJWC.EQ.0) GOTO 45
      CALL BLDOBJ
C
C....  DO NEXT LINE IF NOT END
C
45    IF(ISERR.EQ.1) GOTO 50
      I=JADD(PC,NEWPC,PC)
      GOTO 30
C
C....  END OF ASSEMBLY, OUTPUT BALANCE OF OBJ BUFFER
C
200   ISERR=0
      READ(1,210,END=250) (SRCLNE(I),I=1,80)
210   FORMAT(80A1)
      NOCARD=NOCARD+1
C
C CONVERT ALL CHARACTERS
C
      DO 225 I=1,80
      IF(SRCLNE(I).GE.32) GO TO 220
215   SRCLNE(I)=32
      GO TO 225
220   IF(SRCLNE(I).LT.96) GO TO 225
      SRCLNE(I)=SRCLNE(I)-32
      IF(SRCLNE(I).GE.96) GO TO 215
225   CONTINUE
C
C REMOVE TRAILING BLANKS
C
      LNELEN=80
230   IF(SRCLNE(LNELEN).NE.32) GO TO 240
      LNELEN=LNELEN-1
      IF(LNELEN.GT.0) GO TO 230
240   LNELEN=LNELEN+1
      SRCLNE(LNELEN)=0
      GO TO 500
C

C END OF FILE
C
250   ISERR=1
      GO TO 500
C
C REWIND SOURCE FILE
C
300   REWIND 1
      NOCARD=0
      GO TO 500
C
C CLOSE SOURCE FILE
C
400   CLOSE(UNIT=1)
500   RETURN
      END
      SUBROUTINE LIST(LCODE)
C
C PERFORMS OPEN AND CLOSE ON LIST FILE
C
C INPUT:LCODE = 1 => OPEN FILE (NAME READ FROM KEYBOARD)
C               2 => CLOSE FILE
C
      BYTE FILNAM(12)
      INTEGER PASS
      BYTE NAME(8)
C
      COMMON /LST/ LUNIT,PASS,NAME,NOPAGE,NOLINE,MEFLG,IERCNT
C
      COMMON /FNAM/ FILNAM,OBJFLG
C INPUT:
C      ICODE = 1 => FIND OPERAND IN SYMBOL TABLE.  IF NOT FOUND,
C                   IT IS ENTERED INTO THE TABLE AS REFERENCED
C                   BUT NOT DEFINED.  THE INDEX OF THE SYMBOL
C                   IN THE SYMBOL IS RETURNED IN 'STIND'.
C
C             2 => FIND LABEL IN SYMBOL TABLE.  IF FOUND AND ALREADY
C                   DEFINED AND THIS IS THE FIRST PASS OF THE
C                   ASSEMBLER, THE MULTIPLE DEFINED BIT IS SET IN
C                   SYMFLG.  IF FOUND BUT ONLY PREVIOUSLY REFERENCED,
C                   THE DEFINED BUT PREVIOUSLY REFERENCED BIT IS SET
C                   AND THE REFERENCED BIT IS CLEARED.  IF NOT FOUND,
C                   IT IS ENTERED AND THE DEFINED BIT IS SET.
C
C      IADDR =      ADDRESS OF SYMBOL FOR ENTERING INTO SYMBOL TABLE.
C      SYMBOL=      SYMBOL TO LOOK UP OR ENTER IN SYMBOL TABLE.
C
C OUTPUT:
C      STIND = INDEX INTO SYMBOL TABLE FOR SYMBOL.
C
C FORMAT OF 'SYMFLG':

C
C BIT   MEANING IF SET
C  0    SYMBOL HAS BEEN REFERENCED BUT NOT DEFINED.
C  1    SYMBOL HAS BEEN DEFINED AND WAS REFERENCED BEFORE DEFINITION.
C  2    SYMBOL HAS BEEN DEFINED AND THERE WERE NO REFERENCES BEFORE.
C  3    SYMBOL HAS BEEN MULTIPLE DEFINED.
C  4    SYMBOL IS AN EQUATED VALUE
C
      IMPLICIT INTEGER (A-Z)
      BYTE SYMFLG(513),SYMSTR(8),SRCLNE(81)
      DIMENSION SYMSYM(4,512),SYMBOL(4),SYMLIN(512)
      INTEGER*4 SYMADR(512),IADDR
      INTEGER*4 PC,NEWPC
      COMMON/SYMT/STIND,SYMADR,PC,NOSYM,NEWPC,SYMFLG
      BYTE NAME(8)
      COMMON /LST/ LUNIT,PASS,NAME,NOPAGE,NOLINE,MEFLG,IERCNT
      COMMON /SRC/ LNELEN,ISERR,NOCARD,SRCLNE
      COMMON/SYMN/SYMSYM,SYMLIN
C
C     PACK SYMBOL TWO BYTES TO A WORD
C
      DO 100 J=1,4
      I = J*2
100   SYMBOL(J) = ((SYMSTR(I-1)*256).OR.SYMSTR(I))
C
C     SEARCH FOR SYMBOL IN SYMBOL TABLE
C
      STIND  = 1
      NOVFLG = 0
      IF(NOSYM.EQ.0) GO TO 200
      DO 120 STIND=1,NOSYM
      DO 110 J=1,4
      IF(SYMSYM(J,STIND).NE.SYMBOL(J)) GO TO 115
110   CONTINUE
      GO TO 300
115   DO 118 J=1,4
```

```
C
50      ENDFLG = 1
        CALL BLDOBJ
C
C....   PRINT SYMBOL TABLE
C
        CALL PST
C
C....   CLOSE FILES AND DO IT AGAIN
C
        CALL SOURCE(4)
        CALL LIST(2)
        CALL OBJECT(2)
        GOTO 5
        END
        SUBROUTINE SOURCE(ICODE)
C
C PERFORMS ALL OPERATIONS OF SOURCE INPUT FILE
C
C INPUT:
C ICODE = 1 => OPEN SOURCE FILE (NAME READ FROM KEYBOARD)
C         2 => READ ONE LINE FROM SOURCE FILE INTO
C              'SRCLNE' (80R1 FORMAT), TRAILING BLANKS
C              ARE DELETED, ZERO CHAR IS INSERTED AT
C              THE END OF THE LINE.
C         3 => REWIND SOURCE FILE.
C         4 => CLOSE SOURCE FILE.
C
C OUTPUT:

C       SRCLNE = SOURCE LINE FOR CODE 2
C       LNELEN = LENGTH OF LINE FOR CODE 2
C       ISERR  = 1 IF END OF FILE ON READ (ZERO OTHERWISE)
C       NOCARD = CARD NUMBER READ FROM SOURCE (1-?)
C
        BYTE FILNAM(12)
        BYTE SRCLNE(81)
        COMMON/SRC/LNELEN,ISERR,NOCARD,SRCLNE
        COMMON /FNAM/ FILNAM,OBJFLG
C
C SELECT FUNCTION
C
        GO TO (100,200,300,400),ICODE
C
C OPEN SOURCE FILE
C
100     TYPE 110
110     FORMAT('$Src file name: ')
        READ (5,120) ICNT,FILNAM
120     FORMAT(Q,12A1)
        IF(ICNT.EQ.0) STOP
        CALL ASSIGN(1,FILNAM,ICNT)
        NOCARD=0
        GOTO 500
C
C READ SOURCE LINE
```

(Continued on page 16, column 2)

```
C
C SELECT FUNCTION
C
        GO TO (100,200),LCODE
C
C....   ASSIGN DEFAULT LISTING TO CONSOLE
C
100     LUNIT=5
        TYPE 110
110     FORMAT('$Lst file name: ')
        READ (5,115) ICNT,FILNAM
115     FORMAT(Q,12A1)
        IF(ICNT.EQ.0) GOTO 116
C
C....   IF THERE IS A FILENAME ASSIGN LISTING TO LUN 3
C
        LUNIT = 3
        CALL ASSIGN(LUNIT,FILNAM,ICNT)
116     NOPAGE=0
        GO TO 300
C
C CLOSE FILE
C

200     IF(LUNIT.EQ.5) RETURN
        CALL CLOSE(LUNIT)
300     RETURN
        END
        SUBROUTINE OBJECT(ICODE)
C
C PERFORMS OPEN AND CLOSE ON OBJECT FILE
C
        BYTE FILNAM(12)
        INTEGER PASS
        BYTE NAME(8)
        COMMON /LST/ LUNIT,PASS,NAME,NOPAGE,NOLINE,MEFLG,IERCNT
        COMMON /FNAM/ FILNAM,OBJFLG
        GOTO (100,200),ICODE
100     TYPE 110
110     FORMAT ('$Obj file name: ')
        READ (5,115) ICNT,FILNAM
115     FORMAT(Q,12A1)
        IF(ICNT .EQ.0) GOTO 116
        CALL ASSIGN(2,FILNAM,ICNT)
        OBJFLG = 1
        RETURN
116     OBJFLG = 0
        RETURN
200     IF(OBJFLG.EQ.0) RETURN
        CALL CLOSE(2)
        RETURN
        END
        SUBROUTINE SYMTBL(ICODE,IADDR,SYMSTR)
C
C SYMBOL TABLE PROCESSOR
C
```

(Continued on page 16, column 3)

```
        IF (SYMSYM(J,STIND).LT.SYMBOL(J)) GOTO 120
        IF (SYMSYM(J,STIND).EQ.SYMBOL(J)) GOTO 118
        MOVFLG = 1
        GOTO 200
118     CONTINUE
120     CONTINUE
C
C SYMBOL WAS NOT FOUND
C
200     IF(NOSYM.LT.513) GO TO 210
        CALL ERROR(221)
        STIND=513
        GOTO 400
210     IF (MOVFLG.EQ.0) GOTO 218
        ITEMP = NOSYM
211     DO 212 J=1,4
212     SYMSYM(J,ITEMP+1) = SYMSYM(J,ITEMP)
        CALL JMOV (SYMADR(ITEMP),SYMADR(ITEMP+1))
        SYMFLG(ITEMP+1) = SYMFLG(ITEMP)
        SYMLIN(ITEMP+1) = SYMLIN(ITEMP)
        ITEMP = ITEMP - 1
        IF (ITEMP.GE.STIND) GOTO 211
218     NOSYM = NOSYM + 1
        DO 220 J = 1,4
220     SYMSYM (J,STIND) = SYMBOL(J)
        IF(ICODE.EQ.1) GO TO 250
        SYMFLG(STIND)=4
        CALL I4CLR(SYMADR(STIND))
        I=JADD(SYMADR(STIND),IADDR,SYMADR(STIND))
        SYMLIN(STIND) = NOCARD
        GOTO 400
250     CALL I4CLR(SYMADR(STIND))
        SYMFLG(STIND)=1
        SYMLIN(STIND) = 0
        GOTO 400

C
C SYMBOL FOUND
C
300     IF(PASS.EQ.2.OR.ICODE.EQ.1) GOTO 400
        IF(SYMFLG(STIND).NE.1) GO TO 310
        SYMFLG(STIND)=2
        CALL I4CLR(SYMADR(STIND))
        I=JADD(SYMADR(STIND),IADDR,SYMADR(STIND))
        SYMLIN(STIND) = NOCARD
        GOTO 400
310     SYMFLG(STIND)=SYMFLG(STIND).OR.8
400     RETURN
        END
        SUBROUTINE CNVHEX(INDEX)
C
C CONVERTS 4 BITS TO HEX ASCII AND INSERTS INTO 'PL' AT 'INDEX'
C
C INPUT: WORD = VALUE
C        INDEX= WHERE TO INSERT IN PL
C
```

(Continued on next page)

68000 Cross Assembler (XASM)

(Listing continued, text begins on page 12)

Listing One

```
C OUTPUT:
C        WORD = WORD/16
C
        BYTE PL(132),DIG
        INTEGER WORD
        COMMON /CNVT/ WORD,PL
        CALL GETBIT(WORD,DIG)
        PL(INDEX)=DIG
        RETURN
        END
        SUBROUTINE INSDAT(IPL,IDIG)
C
C CONVERTS BINARY DATA TO HEX ASCII AND INSERTS INTO 'PL'
C
C INPUT:IPL = INDEX TO INSERT INTO PL
C       IDIG= NUMBER OF DIGITS TO CONVERT AND INSERT
C       WORD= VALUE TO CONVERT (IN COMMON - NOT REFERENCED HERE)
C
        I=IDIG
5       J=IPL+I-1
        CALL CNVHEX(J)
        I=I-1
        IF(I.LE.0) RETURN
        GO TO 5
        END

        SUBROUTINE IHX(ISZ,IDTA,IPPOS)
C
C PRINT A 4 OR 8 DIGIT HEX VALUE
C NUMBER OBTAINED STARTING AT 'WORD'
C AND PUT INTO PRINT BUFFER 'PL' STARTING IN COL 1
C
        IMPLICIT INTEGER (A-Z)
        COMMON /CNVT/ WORD,PL
        COMMON /LST/LUNIT,PASS,NAME,NOPAGE,NOLINE,NEFLG,IERCNT
        BYTE PL(132),NAME(8)
        DIMENSION IDTA(3)
        PL(1)=32
        IF(ISZ.EQ.2) GOTO 15
        WORD=IDTA(1)
        CALL INSDAT(IPPOS,4)
        RETURN
15      WORD=IDTA(2)
        CALL INSDAT(IPPOS,4)
        WORD=IDTA(1)
        CALL INSDAT(IPPOS+4,4)
        RETURN
        END
        SUBROUTINE NEWPAG
        IMPLICIT INTEGER (A-Z)
C
C PUTS OUT HEADERS AT TOP OF EACH PAGE
C
        INTEGER PASS
        BYTE NAME(8),FF
        COMMON /LST/ LUNIT,PASS,NAME,NOPAGE,NOLINE,NEFLG,IERCNT
        FF='14
        NOPAGE=NOPAGE+1
        NOLINE = 57
        IF(NOPAGE.EQ.1) FF = 0
        WRITE(LUNIT,10)FF,NAME,NOPAGE
10      FORMAT(' ',1A1,8A1,T28,'M68000 CROSS-ASSEMBLER X1.0
     +',T83,'PAGE ',I3,/)
        RETURN
        END
        SUBROUTINE PAGCHK
        IMPLICIT INTEGER (A-Z)
C
C CHECKS TO SEE IF A PAGE HAS BEEN FILLED
C
        BYTE NAME(8)
        COMMON /LST/ LUNIT,PASS,NAME,NOPAGE,NOLINE,NEFLG,IERCNT
        IF(NOLINE.EQ.0) CALL NEWPAG
        RETURN
        END
        SUBROUTINE ERROR(IERR)
        IMPLICIT INTEGER(A-Z)
C
C       AND PRINTS ERROR MESSAGE DURING PASS 2
C
        COMMON /LST/ LUNIT,PASS,NAME,NOPAGE,NOLINE,NEFLG,IERCNT
        COMMON /SRC/ LNELEN,ISERR,NOCARD,SRCLNE
        COMMON /PRSE/ OPPTR,MODPTR,OPNPTR,LABEL,CMTPTR
     +,PRFLG,SCANPT,OPCLEN,OPNPT2,INODE
        COMMON /OBJOUT/ OBJBUF,OBJWC,LFLG,RFLG,DBFLG
        DIMENSION OBJBUF(40)
        COMMON /SYMT/STIND,SYMADR,PC,NOSYM,NEWPC,SYMFLG
        INTEGER*4 PC,NEWPC,SYMADR(512)
        LOGICAL*1 SYMFLG(513),ERRPTR(80),NAME(8),SRCLNE(81)
        LOGICAL*1 LABEL(8)
C
C....   ERRORS ARE IGNORED DURING THE FIRST PASS
C
        IF(PASS.EQ.1) RETURN
C
        PRFLG  = 3
C
C....   WE NEED AT LEAST THREE LINES TO PRINT AN BAD LINE
C
        IF(NOLINE.LE.2) NOLINE = 0
        CALL PAGCHK
C
C....   IF THIS IS NOT THE FIRST ERROR THEN DON'T PRINT THE LINE
C
        IF (NEFLG.EQ.1) GOTO 15
        WRITE(LUNIT,10) NOCARD,(SRCLNE(I),I=1,LNELEN-1)
C
        LSWRDS = OBJWC
        IF(OBJWC.GT.5) LSWRDS=5
C
C       CHECK IF WE HAVE TO GO TO NEXT PAGE
C
        CALL PAGCHK
C
C


        IF(CMTPTR.NE.1)GOTO 80
        OPPTR=1
        GOTO 220
80      GOTO (200,200,200,410,500,600,200,200,200,400),PRFLG+1
200     CALL IHX(2,PC,7)
C
C
        IF(LSWRDS.EQ.0) GOTO 212
205     DO 210,I=1,LSWRDS
210     CALL IHX(1,OBJBUF(I),11+(5*I))
C
C
212     IF(LABEL(1).EQ.0) GOTO 220
        DO 215,I=1,8
215     PL(I+40)=LABEL(I)
220     J=0
        DO 230 I=OPPTR,LNELEN
        PL(J+50)=SRCLNE(I)
        IF(SRCLNE(I).EQ.'40) GOTO 240
230     J=J+1
        GOTO 1000
240     III=0
        DO 250 II=I+1,LNELEN
        IF (II.EQ.CMTPTR) III = 25
        PL(57+III)=SRCLNE(II)
        III = III + 1
250     IF ((III + 57).GT.132) GOTO 255
        GOTO 1000
255     PL(132) = 0
        GOTO 1000
C
C       PRFLG = 3  (NEW PAGE)
C
400     CALL NEWPAG
410     RETURN
C
C
500     GOTO 205
C
C
600     GOTO 220
C
C
700     CALL IHX(2,OBJBUF(2),16)
        GOTO 212
C
C
1000    DO 1001 I=48,132
```

```
      SUBROUTINE PST
C
C SORT AND PRINT SYMBOL TABLE
C
      INTEGER PASS,STIND,SYMLIN(512)
      INTEGER*4 PC,NEWPC,SYMADR(512)
      BYTE NAME(8),SYMSYM(8,512),SYMFLG(513),PL(132)
      COMMON/LST/LUNIT,PASS,NAME,NOPAGE,NOLINE,MEFLG,IERCNT
      COMMON/SYMN/SYMSYM,SYMLIN
      COMMON/SYMT/STIND,SYMADR,PC,NOSYM,NEWPC,SYMFLG
      COMMON /CNVT/ WORD,PL
      IF(NOSYM.EQ.0) RETURN
C
C     START OUT WITH CLEAN BUFFER
C
      DO 50 I = 1,132
50    PL(I) = '40
C
C GOTO TOP OF PAGE
C
      CALL NEWPAG
C
C     GENERATE THE SYMBOL LIST A LINE AT A TIME
C
      DO 300 I = 1,NOSYM,5
      DO 210 IDX=0,4
      IF (I+IDX.GT.NOSYM) GOTO 290
      DO 170 IPT=1,7,2
      PL(IPT+(IDX*24)+1) = SYMSYM(IPT,(I+IDX))
170   PL(IPT+(IDX*24)) = SYMSYM(IPT+1,(I+IDX))
      CALL IHX(2,SYMADR(I+IDX),(IDX*24)+12)
      IFTMP = SYMFLG(I+IDX)
      IF ((IFTMP.AND.16).NE.16 ) GOTO 180
      PL((IDX*24)+19) = 'E'
      PL((IDX*24)+20) = 'Q'
180   IF ((IFTMP.AND.8).NE.8  ) GOTO 190
      PL((IDX*24)+19) = 'M'
      PL((IDX*24)+20) = 'U'
190   IF ((IFTMP.AND.1).NE.1) GOTO 200
      PL((IDX*24)+19) = 'U'
      PL((IDX*24)+20) = 'N'
200   IF ((IFTMP.AND.'31).NE.0) GOTO 210
      PL((IDX*24)+19) = ' '
      PL((IDX*24)+20) = ' '
210   CONTINUE
290   WRITE (LUNIT,400) (PL(N),N=1,IDX*24)
      NOLINE = NOLINE -1
      CALL PAGCHK
300   CONTINUE
400   FORMAT (' ',132A1)
      WRITE (LUNIT,410) NOSYM,IERCNT
410   FORMAT (/,' ',I3,' SYMBOLS , ',I3,' ERRORS DETECTED')
      IF (LUNIT.EQ.5) RETURN
      WRITE (5,410)     NOSYM,IERCNT
      RETURN
      END
```

(Continued on page 18, column 2)

```
10    FORMAT(' ',/,' ',I4,35X,80A1:)
      NOLINE = NOLINE - 2
15    DO 20,I=1,SCANPT
20    ERRPTR(I)='40
      ERRPTR(I)='136
      WRITE(LUNIT,30) IERR,(ERRPTR(I),I=1,SCANPT+1)
30    FORMAT(' ++++  ERROR    ',I3,20X,80A1:)
      NOLINE = NOLINE - 1
      IERCNT = IERCNT + 1
      MEFLG = 1
      RETURN
      END
      SUBROUTINE LSTLNE
      IMPLICIT INTEGER (A-Z)
C
C     BUILD LINE (OR LINES IF DC.B DC.W DC.L)
C     FOR DISPLAY
C
      COMMON /SRC/ LNELEN,ISERR,NOCARD,SRCLNE
C
      COMMON /CNVT/ WORD,PL
C
      COMMON /LST/ LUNIT,PASS,NAME,NOPAGE,NOLINE,MEFLG,IERCNT
C
      COMMON /OBJOUT/ OBJBUF,OBJWC,LFLG,RFLG,DBFLG
C
      COMMON /SYMT/ STIND,SYMADR,PC,NOSYM,NEWPC,SYMFLG
C
      COMMON /PRSE/ OPPTR,MODPTR,OPMPTR,LABEL,CMTPTR
     +,PRFLG,SCANPT,OPCLEN,OPMPT2,IMODE
C
      INTEGER*4 PC,NEWPC,SYMADR(512)
      DIMENSION OBJBUF(40)
      BYTE SYMFLG(513),NAME(8),LABEL(8),SRCLNE(81),PL(132)
      DATA PL/132*'40/
C
C     PRFLG = 0    ERRORS DETECTED (PRINT LINE AS READ)
C             1    NO ERRORS DETECTED (PRINT NORMALLY)
C             2    DC.W  / DC.L DIRECTIVES
C             3    SUPRESS PRINTOUT OF LINE
C             4    DC.B DIRECTIVE
C             5    NAM / END / MON DIRECTIVES
C             6    EQU / SET DIRECTIVES
C             7    ORG / RORG DIRECTIVES
C             8    DS   DIRECTIVE
C             9    PAGE DIRECTIVE
C
C
C
C     IF THIS IS THE FIRST PASS, THEN DONT PRINT ANYTHING
C
      IF (PASS.EQ.1) RETURN
C
C     IF CODE IS LONGER THAN FIVE WORDS THEN
C     ONLY PRINT 5 WORDS OF AN INSTRUCTION
```

(Continued on page 18, column 3)

```
1001  IF(PL(I).EQ.0)GOTO 1002
1002  WRITE(LUNIT,1110) NOCARD,(PL(II),II=6,I-1)
1110  FORMAT(' ',I4,132A1)
      DO 1120 II = 1,I
1120  PL(II) = '40
      NOLINE = NOLINE - 1
      RETURN
      END

      SUBROUTINE BLDOBJ
      IMPLICIT INTEGER (A-Z)
C
C     BUILD OBJ FILE
C
      COMMON /FNAM  / FILNAM,OBJFLG
C
      COMMON /OBJOUT/ OBJBUF,OBJWC,LFLG,RFLG,DBFLG
C
      COMMON /CNVT  / WORD,PL
C
      COMMON /SYMT  / STIND,SYMADR,PC,NOSYM,NEWPC,SYMFLG
C
      COMMON /HEXFLG/ ENDFLG,HEXWC,HEXPC,OLDPC
C
      DIMENSION OBJBUF(40),HEXBUF(8)
      INTEGER*4 PC,NEWPC,SYMADR(512),OLDPC,NEWVAL,HEXPC
      LOGICAL*1 SYMFLG(513),PL(132),FILNAM(12)
C
C     CHECK IF OBJ FILE IS TO BE GENERATED
C
      IF (OBJFLG.EQ.0) RETURN
C
C     CHECK FOR THE END OF ASSEMBLY FLAG
C     IF IT IS SET, WRITE OUT THE BALANCE OF THE OBJ BUFFER
C
      IF (ENDFLG.EQ.0) GOTO 10
      IF (HEXWC.NE.0)  CALL WRTOBJ(HEXPC,HEXWC,HEXBUF)
      RETURN
C
C     CHECK THE CURRENT VALUE OF THE PC WITH THAT OF THE ONE SAVED
C     IF THE TWO ARE NOT EQUAL, THEN WRITE OUT THE BALANCE OF THE
C     OBJ BUFFER AND START AT THE NEW PC VAL
C
10    CALL DBLSGL(PC,PC1,PC2)
      CALL DBLSGL(OLDPC,OLDPC1,OLDPC2)
      IF (PC1.NE.OLDPC1) GOTO 50
      IF (PC2.EQ.OLDPC2) GOTO 75
50    IF (HEXWC.NE.0) CALL WRTOBJ(HEXPC,HEXWC,HEXBUF)
      CALL JMOV(PC,HEXPC)
      CALL JMOV(PC,OLDPC)
C
C     EXTRACT OBJECT WORDS FROM OBJECT BUFFER AND
C     PUT THEM INTO AN INTERNAL BUFFER. IF THE
C     INTERNAL BUFFER IS FULL, THEN OUTPUT THE BUFFER.
C
```

(Continued on next page)

68000 Cross Assembler (XASM)

(Listing continued, text begins on page 12)

Listing One

```
75      I = 1
76      HEXWC = HEXWC + 1
        HEXBUF(HEXWC) = OBJBUF(I)
        IF (HEXWC.NE.8) GOTO 99
C....   OBJECT BUFFER IS FULL, OUTPUT IT TO OBJ FILE
C
        CALL WRTOBJ(HEXPC,HEXWC,HEXBUF)
C
C       CALCULATE NEW STARTING PC FOR HEX BUFFER
C
        N = JICVT(I*2,NEWVAL)
        N = JADD(PC,NEWVAL,HEXPC)
99      I = I + 1
        IF (I.LE.OBJWC) GOTO 76
C
C       CALCULATE WHAT THE NEW PC SHOULD BE BY ADDING
C       THE OBJECT WORD COUNT TO THE CURRENT PC
C
        I = JADD(OLDPC,NEWPC,OLDPC)
        RETURN
        END
        SUBROUTINE WRTOBJ(HEXPC,HEXWC,HEXBUF)
        IMPLICIT INTEGER(A-Z)
C
C       OUTPUT THE CONTENTS OF THE OBJECT BUFFER
C
C       HEXPC = STARTING PC FOR BUFFER
C       HEXWC = NUMBER OF WORDS USED IN BUFFER
C       HEXBUF= 8 WORD OBJECT BUFFER
C
        COMMON /CNVT/ WORD,PL
        LOGICAL*1 PL(132)
        INTEGER*4 HEXPC
        DIMENSION HEXBUF(8)
        DO 10, I = 1,80
10      PL(I) = '40
        CALL IHX(2,HEXPC,1)
        PLIDX = 10
        DO 20,I=1,HEXWC
20      CALL IHX(1,HEXBUF(I),PLIDX+(5*(I-1)))
        WRITE (2,100)(PL(I),I=3,10+(5*HEXWC))
100     FORMAT(' ',80A1)
        HEXWC = 0
        DO 900, I = 1,80
900     PL(I) = '40
        RETURN
        END
        SUBROUTINE PRCESS
C
        COMMON /OPWD  / OPMFLG,OPNWC,OPNWRD
C
        COMMON /LST   / LUNIT,PASS,NAME,NOPAGE,NOLINE,MEFLG,IERCNT
C
        COMMON /OBJOUT/ OBJBUF,OBJWC,LFLG,RFLG,BRFLG
C
        COMMON /SRC   / LNELEN,ISERR,NOCARD,SRCLNE
C
        COMMON /SYMT  / STIND,SYMADR,PC,NOSYM,NEWPC,SYMFLG
C
        COMMON /PRSE  / OPPTR,MODPTR,OPMPTR,LABEL,CNTPTR
     +,PRFLG,SCANPT,OPCLEN,OPMPT2,IMODE
C
        COMMON /OPCPTS/ OPTYP,OPSKEL,OPSK2,OPIDX
C
        INTEGER*4 PC,NEWPC,SYMADR(512),SYMVAL,TMPVAL,J2
        LOGICAL*1 SRCLNE(81),LABEL(8),NAME(8),SYMFLG(513)
        DIMENSION OBJBUF(40),OPNWRD(3)
C
C....   SET UP FLAGS THAT CHANGE EACH TIME THRU
C
        CALL I4CLR(NEWPC)
        J2     = 2
        OP1EA  = 0
        OP2EA  = 0
        OP1DA  = 0
        OP2DA  = 0
        OPNWC = 0
C
C....   DECODE OPCODE
C
        CALL DECOPC
        IF(OPTYP.NE.0) GOTO 10
        CALL ERROR(400)
        RETURN
C
C....   SKIP IF NO OPERANDS
C
10      IF(OPMPTR.EQ.0)GOTO 20
C
C....   DECODE FIRST OPERAND
C
        OP1EA=OPMPTR
        CALL EATYP(OP1EA,OP1DA)
        IF(OPMPT2.EQ.0)GOTO 20
C
C....   DECODE SECOND OPERAND
C
        OP2EA=OPMPT2
        CALL EATYP(OP2EA,OP2DA)

C
C....   CHECK FOR OPERANDS
C
20      IF(OP1EA.EQ.10.OR.OP2EA.EQ.10) GOTO 8500
        IF(OPTYP.EQ.1.OR.OPTYP.EQ.2) GOTO 90
        IF(OPMPTR.NE.0) GOTO 90
        CALL ERROR(401)
        IF(IMODE.NE.3) GOTO 125
        I=JICVT(4,NEWPC)
        I=JMUL(NEWPC,OPNWRD(2),NEWPC)
        GOTO 128
122     I=JMOV(OPNWRD(2),NEWPC)
        GOTO 128
125     I=JICVT(2,NEWPC)
        I=JMUL(NEWPC,OPNWRD(2),NEWPC)
128     I=JMOV(PC,OBJBUF(2))
        I=JMOV(PC,SYMVAL)
        GOTO 7005
C
C       ORG
C
130     IF(LABEL(1).EQ.0) GOTO 132
131     CALL ERROR(402)
        RETURN
C
132     RFLG=1
133     PRFLG=7
        IF(OPMPTR.NE.0) GOTO 134
        CALL I4CLR(NEWPC)
        CALL I4CLR(PC)
        RETURN
134     CALL PROCOP(OPMPTR)
        IF(OPNWC.EQ.7) GOTO 135
        CALL I4CLR(PC)
        I=JADD(NEWPC,OPNWRD(2),NEWPC)
        RETURN
135     CALL ERROR(403)
        RETURN
C
C       END <STARTING ADR>
C
140     ISERR=1
        IF(LABEL(1).NE.0) GOTO 131
        PRFLG=5
        RETURN
C
C       EQU
C
150     IF(LABEL(1).EQ.0) GOTO 131
        PRFLG=6
        IF(OPMPTR.EQ.0) GOTO 8500
        CALL PROCOP(OPMPTR)
        IF(OPNWC.EQ.7) RETURN
        CALL SYMTBL(2,OPNWRD(2),LABEL)
        IF((SYMFLG(STIND).AND.'10).EQ.'10)CALL ERROR(409)
        I=JMOV(OPNWRD(2),SYMADR(STIND))
        SYMFLG(STIND)=SYMFLG(STIND).OR.16
        I=JMOV(SYMADR(STIND),OBJBUF(2))
        RETURN
C
C       RORG
C
160     IF(LABEL(1).NE.0) GOTO 131
        RFLG=0
        GOTO 133
```

```
C       PROCESSES SOURCE LINE AFTER IT HAS BEEN PARSED BY PARSE
C
C INPUT:PARSE OUTPUTS
C
C OUTPUT:
C
C  OBJWC    NUMBER OF WORDS REQUIRED FOR INSTRUCTION
C
C  OBJBUF   TABLE OF WORDS GENERATED
C
C  PRFLG 0  ERRORS DETECTED (PRINT LINE AS READ
C        1  NO ERRORS DETECTED (PRINT NORMALLY)
C        2  DC.W/DC.L DIRECTIVES
C        3  DONT PRINT LINE
C        4  DC.B DIRECTIVE
C        5  NAM/END/MOM DIRECTIVES
C        6  EQU/SET DIRECTIVE
C        7  ORG/RORG DIRECTIVE
C        8  DS DIRECTIVE
C        9  PAGE DIRECTIVE
C
C  NEWPC    NEW VALUE FOR PC
C
C
C  OP1EA 0  NOT REG OR IMMEDIATE DATA
C        1  D REG
C        2  A REG
C        3  (AN)
C        4  (AN)+
C        5  -(AN)
C        6    DATA
C        7  SR
C        8  CCR
C        9  USP
C        10 ERROR DETECTED
C
C  IMODE 0  NO SIZE SPECIFIED (DEFAULT IS WORD)
C        1  .B
C        2  .W
C        3  .L
C        4  .S  (SHORT BRANCH)
C
C       ERRORS DEFINED.....
C
C       400     UNDEFINED OPCODE
C       401     OPERAND MISSING FOR OPCODE
C       402     NO ORG SPECIFIED FOR ORG INSTRUCTION
C       403     ERROR IN DC OPN VALUE
C       406     GENERAL ERROR IN DECODING
C       407     UNDEFINED SYMBOL
C       408     ERROR IN SIZE OF Y(Ax,Rx) INDEX
C       409     MULT DEFN SYMBOL
C
        IMPLICIT INTEGER (A-Z)
C
```

(Continued on page 20, column 2)

```
        RETURN
C
C....   DEFAULT SIZE IS ONE WORD FOR INSTRUCTIONS
C
90      OBJWC=1
C
C ++++++++++++++++++++++++++++++++++++++++++++++
C
C GOTO OPCODE EVALUATION ROUTINES VIA OPTYPE
C
C ++++++++++++++++++++++++++++++++++++++++++++++
        GOTO(100,200,300,500,400,600,700,800,900,1000
     +,1100,1200,1300,1400,1500,1600,1700,1800,1900,2000,2100),OPTYP
C
C ++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS PSEUDO OPS
C
C ++++++++++++++++++++++++++++++++++++++++++++++
C
C....   PSEUDO OPS NORMALLY DON'T GENERATE CODE
C....   THE EXECEPTION BEING 'DC'
C
100     OBJWC=0
        GOTO(110,120,130,140,150,195,150,160,170,180,190),OPIDX
C
C       DC
C
110     PRFLG=2
        IFLG = RFLG
        RFLG = 1
111     CALL PROCOP(OPNPTR)
        IF(OPNWC.EQ.0) GOTO 115
        IF(IMODE.EQ.3) OBJWC = OBJWC+2
        IF(IMODE.NE.3) OBJWC = OBJWC+1
        IF(IMODE.EQ.3) OBJBUF(OBJWC-1) = OPNWRD(3)
        IF(IMODE.EQ.3) OBJBUF(OBJWC  ) = OPNWRD(2)
        IF(IMODE.NE.3) OBJBUF(OBJWC  ) = OPNWRD(2)
        IF(SRCLNE(OPNPTR).NE.'54) GOTO 119
        OPNPTR = OPNPTR+1
        GOTO 111
115     CALL ERROR(403)
118     RFLG = IFLG
        GOTO 7000
119     IF(IMODE.NE.1.OR.OPNWRD(2).GE.256) GOTO 118
        OBJBUF(OBJWC)=(OBJBUF(OBJWC)$'400)
        GOTO 118
C
C       DS
C
120     PRFLG=7
        IF(OPNPTR.EQ.0) GOTO 8500
        CALL PROCOP(OPNPTR)
        IF(OPNWC.EQ.7) GOTO 134
        IF(IMODE.EQ.1) GOTO 122
```

(Continued on page 20, column 3)

```
C
C       PAGE
C
170     IF(LABEL(1).NE.0)GOTO 131
        LFLG=0
        PRFLG=9
        RETURN
C
C       LIST
C
180     IF(LABEL(1).NE.0)GOTO 131
        LFLG=1
        PRFLG=3
        RETURN
C
C       NLIST
C
190     IF(LABEL(1).NE.0) GOTO 131
        LFLG=0
        PRFLG=3
        RETURN
C
C       NAM
C
195     IF(LABEL(1).NE.0) GOTO 131
        DO 197 I=1,8
197     NAME(I)='40
        N=1
        DO 196 I=OPNPTR,OPNPTR+7
        NAME(N)=SRCLNE(I)
        N=N+1
196     IF(I.EQ.LNELEN-1) RETURN
        RETURN
C ++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS INHERENT INSTRUCTIONS..IE NOP
C
C ++++++++++++++++++++++++++++++++++++++++++++++
200     OBJBUF(1)=OPSKEL
        GOTO 7000
C ++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS MOVE INSTRUCTION
C       <EA>,<EA>  SR,<EA>  <EA>,CCR  <EA>,SR  USP,An  An,USP
C
C ++++++++++++++++++++++++++++++++++++++++++++++
C
C....   LOOK FOR OBVIOUS MISTAKES
C
300     IF(OP2EA .EQ.6.OR.OP1EA .EQ.8) GOTO 8500
        IF(OP1EA.EQ.9 .AND.OP2EA.NE.2) GOTO 8500
        IF(OP1EA.NE.2 .AND.OP2EA.EQ.9) GOTO 8500
        IF(OPNPTR.EQ.0.OR.OPNPT2.EQ.0) GOTO 8500
C
C....   SR,<EA> - USP,<EA>
```

(Continued on next page)

68000 Cross Assembler (XASM)

(Listing continued, text begins on page 12)

Listing One

```
C
        IF(OP1EA.EQ.7.OR. OP1EA.EQ.9) GOTO 350
C
C....   OP1EA = 1 THRU 5
C
        IF((OP1EA.GE.1).AND.(OP1EA.LE.5)) GOTO 305
C
C....   PROCESS FIRST OPN HERE IF COMPLEX
C
        CALL PROCOP(OPNPTR)
C
C....   CHECK FOR EA TYPES 7-9
C
303     IF(OP2EA.GT.6) GOTO 340
C
C....   CHECK FOR FIRST OPERAND IMMEDIATE MODE ADDRESSING
C
        IF (OP1EA.NE.6) GOTO 304
C
C....   SKIP MOVQ IF FWD REF SYMBOL
C
        IF(OPNFLG.EQ.1) GOTO 304              ! CANNOT BE FWD REF SYM
        IF(IMODE .NE.3) GOTO 304              ! MUST BE .L MODE
        IF(OPNWRD(3).EQ. 0) GOTO 301          ! HI WORD MUST BE ZERO
        IF(OPNWRD(3).EQ.-1) GOTO 301          ! OR MINUS ONE
        GOTO 304
C
C....   CHECK IF VAL WITHIN RANGE FOR MOVEQ (+/- 128)
C....   ALSO CHECK IF DESTINATION IS A DATA REGISTER
C
301     I=ICKVAL(OPNWRD(2))
        IF ((I.EQ.0).AND.(OP2EA.EQ.1)) GOTO 330
C
C....   ADD IN OPCODE SIZE BITS
C
304     OBJBUF(1)=OBJBUF(1).OR.'30000
        IF(IMODE.EQ.1) OBJBUF(1)=(OBJBUF(1)).AND.'17777
        IF(IMODE.EQ.3) OBJBUF(1)=(OBJBUF(1)).AND.'27777
C
C....   MOVE IN NUMBERS FOR 1ST AND 2ND EXT WORDS
C
        OBJWC     = OBJWC+OPNWC
        OBJBUF(2) = OPNWRD(2)
        OBJBUF(1) = OPNWRD(1)
        IF(OPNWC.EQ.2) OBJBUF(2) = OPNWRD(3)
        IF(OPNWC.EQ.2) OBJBUF(3) = OPNWRD(2)
        GOTO 310

C
C....   PROCESS EA TYPES 0-5 FOR FIRST OPN
        GOTO 7000
C
C....   HANDLE STUFF FOR EA'S 0 AND 6
C
349     OBJBUF(1)=OBJBUF(1).OR.OPNWRD(1)
        OBJBUF(2)=OPNWRD(2)
        IF(OPNWC.EQ.2)OBJBUF(2)=OPNWRD(3)
        IF(OPNWC.EQ.2)OBJBUF(3)=OPNWRD(2)
        OBJWC=OBJWC+OPNWC
        GOTO 7000
C
C....   GENERATE MOVE SR,<EA> - USP,AN
C
350     IF (OP1EA.EQ.9) GOTO 355              ! SR,<EA>
        IF (OP2EA.EQ.2) GOTO 8500             ! USP,AN
        IF (OP2EA.EQ.0) GOTO 353
        OBJBUF(1) = '40300.OR.OP2DA.OR.((OP2EA-1)*'10)
        GOTO 7000
C
353     CALL PROCOP(OPNPT2)
        OBJBUF(2)=OPNWRD(2)
        IF(OPNWC.EQ.2) OBJBUF(2) = OPNWRD(3)
        IF(OPNWC.EQ.2) OBJBUF(3) = OPNWRD(2)
        OBJWC = OBJWC + OPNWC
        OBJBUF(1) = '43000.OR.OPNWRD(1)
        GOTO 7000
C
355     OBJBUF(1) = '47150.OR.OP2DA
        GOTO 7000
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS CMP INSTRUCTION
C       <EA>,DN <EA>,AN  DATA,<EA> (AY)+,(AX)+
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
400     IF((OP1EA.EQ.6).AND.(OP2EA.NE.2)) GOTO 460  ! CMPI INSTR
        IF((OP1EA.EQ.5).AND.(OP2EA.EQ.5)) GOTO 480  ! CMPM INSTR
        IF((OP2EA.EQ.1).OR. (OP2EA.EQ.2)) GOTO 410  ! CMP <EA>,DN OR AN
        GOTO 8500                                   ! ALL ELSE ILLEGAL
C
C....   PROCESS <EA>,DN <EA>,AN
C
410     IF(OP2EA.EQ.2.AND.IMODE.EQ.1) GOTO 8500     ! CMPA CANT HAVE .B
        IF(OP2EA.NE.2) GOTO 411
        IF(IMODE.EQ.3) OPSKEL = OPSKEL.OR.'500          ! CMPA.L
        IF(IMODE.NE.3) OPSKEL = OPSKEL.OR.'200          ! CMPA.W
411     IF((OP1EA.EQ.0).OR.(OP1EA.EQ.6)) GOTO 415       ! COMPLEX OPN
C
C....   PROCESS FOR REG OPNS
C
412     OBJBUF(1)=OPSKEL.OR.(OP2DA*'1000).OR.((OP1EA-1)*'10).OR.OP1DA
        GOTO 6000
C
C....   PROCESS FOR COMPLEX 1ST OPNS
C
C
C....   GENERATE DN,<EA>
C
        OPSKEL = OPSKEL.OR.(OP1DA*'1000)
        IF(OP2EA.EQ.0) GOTO 511
        OBJBUF(1) = OPSKEL.OR.((OP2EA-1)*'10).OR.OP2DA
        GOTO 6000
C
511     CALL PROCOP(OPNPT2)
514     OBJBUF(2) = OPNWRD(2)
        IF(OPNWC.EQ.2) OBJBUF(2) = OPNWRD(3)
        IF(OPNWC.EQ.2) OBJBUF(3) = OPNWRD(2)
        OBJWC = OBJWC+OPNWC
        OBJBUF(1) = OPSKEL.OR.OPNWRD(1)
        GOTO 6000
C
C....   GENERATE <EA>,DN
C
520     OPSKEL = OPSKEL.OR.(OP2DA*'1000)
        IF(OP1EA.EQ.0) GOTO 522
521     OBJBUF(1) = OPSKEL.OR.((OP1EA-1)*'10).OR.OP1DA
        GOTO 6000
C
522     CALL PROCOP(OPNPTR)
        GOTO 514
C
C....   GENERATE <EA>,AN
C
525     IF (IMODE.EQ.1) GOTO 8500
        IF (IMODE.EQ.3) OPSKEL = OPSKEL .OR. '500
        IF ((IMODE.EQ.2).OR.(IMODE.EQ.0)) OPSKEL = OPSKEL.OR.'200
        OPSKEL = OPSKEL .OR.(OP2DA*'1000)
        IF((OP1EA.EQ.0).OR.(OP1EA.EQ.6)) GOTO 522
        OBJBUF(1) = OPSKEL.OR.((OP1EA-1)*'10).OR.OP1DA
        GOTO 6000
C
C....   GENERATE xxxI
C
530     IF(OP2EA.GT.6) GOTO 8500

C
C....   EVALUATE IMMEDIATE EXPRESSION
C
        CALL PROCOP(OPNPTR)
C
C....   TRY GENERATING SHORT FORM OF INSTRUCTION
C....   AFTER CHECKING TO SEE IF OPERAND WAS FWD REF
C
        IF(OPNFLG.EQ.1) GOTO 536
        IF(OPNWRD(2).GE.1.AND.OPNWRD(2).LE.8) GOTO 550
C
C....   GENERATE EXTENSION WORDS
C....   LENGTH OF OPERAND DEPENDS ON THE IMODE OF INSTRUCTION
C
536     OBJBUF(2) = OPNWRD(2)
        IF(OPNWC.EQ.2)OBJBUF(2) = OPNWRD(3)
        IF(OPNWC.EQ.2)OBJBUF(3) = OPNWRD(2)
537     OBJWC = OBJWC + OPNWC
```

```
C
305     OBJBUF(1)=((((OP1EA-1)*'10),OR,OP1DA)
C
C....   CHK FOR SIMPLE SECOND OPERANDS
C
310     IF(OP2EA,EQ,0) GOTO 315
C
C....   CHK FOR SR,CCR,USP
C
        IF(OP2EA,GT,6) GOTO 340
        GOTO 320
C
C....   CALCULATE COMPLEX SECOND OPN
C
315     CALL PROCOP(OPNPT2)
        OBJBUF(OBJWC+1)=OPNWRD(2)
        IF(OPNWC,EQ,2) OBJBUF(OBJWC+2)=OPNWRD(2)
        IF(OPNWC,EQ,2) OBJBUF(OBJWC+1)=OPNWRD(3)
        OBJWC=OBJWC+OPNWC
        I=(OPNWRD(1),AND,'7)*'10
        J=(OPNWRD(1),AND,'70)/8
        OBJBUF(1)=OBJBUF(1),OR,((I+J)*'100),OR,'30000
        GOTO 325
C
C....   PROCESS EA TYPES 0-5 FOR SECOND OPN
C
320     OBJBUF(1)=OBJBUF(1)+((((OP2EA-1),OR,(OP2DA*'10))*'100),OR,'30000
C
C....   ADD IN SIZE BITS
C
325     IF(IMODE,EQ,1)OBJBUF(1)=OBJBUF(1),AND,'17777
        IF(IMODE,EQ,3)OBJBUF(1)=OBJBUF(1),AND,'27777
        GOTO 7000
C
C....   GEN MOVEQ ALSO CLR SIZE BITS IF SET
C
330     OBJBUF(1) = 0
        OBJBUF(1) = (OPNWRD(2),AND,'377),OR,'70000,OR,(OP2DA*'1000)
        GOTO 7000
C
C....   GENERATE MOVE <EA>,SR - <EA>,CCR - AN,USP
C
340     IF(OP2EA,EQ,7) OBJBUF(1)='43300
        IF(OP2EA,EQ,8) OBJBUF(1)='42300
        IF(OP2EA,NE,9) GOTO 342
        OBJBUF(1) = '47140,OR,OP1DA
        GOTO 7000
C

C....   GET NON-REG EA'S IF 0 OR 6
C
342     IF(OP1EA,EQ,0,OR,OP1EA,EQ,6) GOTO 349
C
C....   ELSE JUST ADD OR IN THE EA AND REG
C
        OBJBUF(1)=OBJBUF(1),OR,OP1DA,OR,((OP1EA-1)*'10)
```

(Continued on page 22, column 2)

```
415     CALL PROCOP(OPNPTR)
        OBJBUF(1) = OPSKEL,OR,(OP2DA*'1000),OR,(OPNWRD(1),AND,'77)
        OBJBUF(2)=OPNWRD(2)
        IF(OPNWC,EQ,2) OBJBUF(2)=OPNWRD(3)
        IF(OPNWC,EQ,2) OBJBUF(3)=OPNWRD(2)
        OBJWC=OBJWC+OPNWC
        GOTO 6000
C
C....   CMPI INSTRUCTION
C....   EVALUATE THE IMMEDIATE PART
C
460     CALL PROCOP(OPNPTR)
        OBJWC = OBJWC + OPNWC
        OBJBUF(2)=OPNWRD(2)
        IF(OPNWC,EQ,2) OBJBUF(2) = OPNWRD(3)     ! PLAY GAMES IF 2 WDS
        IF(OPNWC,EQ,2) OBJBUF(3) = OPNWRD(2)
C
C....   CHECK FOR SIMPLE DESTINATION EA
C
        IF((OP2EA,GT,0),AND,(OP2EA,LT,6)) GOTO 470
        IF(OP2EA,GT,6) GOTO 8500
        CALL PROCOP(OPNPT2)
        OBJBUF(1) = OPSK2,OR,(OPNWRD(1),AND,'77)
        OBJBUF(OBJWC+1) = OPNWRD(2)
        IF (OPNWC,EQ,2) OBJBUF(OBJWC+1) = OPNWRD(3)
        IF (OPNWC,EQ,2) OBJBUF(OBJWC+2) = OPNWRD(2)
        OBJWC = OBJWC+OPNWC
        GOTO 6000
C
C....   SECOND EA IS NOT COMPLEX
C
470     OBJBUF(1) = OPSK2,OR,OP2DA,OR,((OP2EA-1)*'10)
        GOTO 6000
C
C....   CMPM (AY)+,(AX)+
C
480     OBJBUF(1)=OPSKEL+((OP2DA*'1000)+OP1DA)
        GOTO 6000
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS ADD,SUB INSTRUCTIONS
C       <EA>,DN <EA>,AN  DN,<EA>  DATA,<EA>
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
500     IF(OP2EA,EQ,2) GOTO 525         ! ADDA,SUBA
        IF(OP1EA,EQ,6) GOTO 530         ! ADDI,SUBI
        IF(OP1EA,EQ,1,OR,OP2EA,EQ,1)  GOTO 510
        GOTO 8500                       ! ALL OTHERS ILLEGAL
C
C....
C
510     IF(OP2EA,EQ,1) GOTO 520
        OPSKEL = OPSKEL ,OR, '400
```

(Continued on page 22, column 3)

```
C
C....   IF DEST THRU REG EVAL IT HERE
C
538     IF(OP2EA,EQ,0) GOTO 540
        OBJBUF(1)=OPSK2,OR,((OP2EA-1)*'10),OR,OP2DA
        GOTO 6000
C
C....   EVAL NON-REG DEST
C
540     CALL PROCOP(OPNPT2)
        OBJWC = OBJWC + OPNWC
        OBJBUF(1) = OPSK2,OR,OPNWRD(1)
        IF(OPNWC,EQ,1) OBJBUF(OBJWC  ) = OPNWRD(2)
        IF(OPNWC,EQ,2) OBJBUF(OBJWC+1) = OPNWRD(3)
        IF(OPNWC,EQ,2) OBJBUF(OBJWC  ) = OPNWRD(2)
        GOTO 6000
C
C....   GENERATE xxxQ
C
550     IF(OPNWRD(2),EQ,8) OPNWRD(2) = 0
        IF(OPSK2,EQ,'2000) OPSK2 = '50400
        IF(OPSK2,EQ,'3000) OPSK2 = '50000
        OPSK2 = OPSK2,OR,(OPNWRD(2)*'1000)
        GOTO 538
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS AND,OR INSTRUCTIONS
C       <EA>,DN  DN,<EA>   DATA,<EA>
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
600     IF(OP1EA,EQ,6) GOTO 610
        IF(OP2EA,NE,1) GOTO 620
C
C....   PROCESS <EA>,DN
C
        OPSKEL=OPSKEL+(OP2DA*'1000)
        IF(OP1EA,EQ,0) GOTO 605
        OBJBUF(1)=OPSKEL,OR,OP1DA,OR,((OP1EA-1)*'10)
        GOTO 6000
C
605     CALL PROCOP(OPNPTR)
        OBJBUF(1)=OPSKEL,OR,OPNWRD(1)
        OBJBUF(2)=OPNWRD(2)
        IF(OPNWC,EQ,2) OBJBUF(2)=OPNWRD(3)
        IF(OPNWC,EQ,2) OBJBUF(3)=OPNWRD(2)
        OBJWC=OBJWC+OPNWC
        GOTO 6000
C
C....   PROCESS  DATA,<EA>
C
610     OPSKEL = OPSK2
        IF(OP2EA,EQ,6) GOTO 8500
        CALL PROCOP(OPNPTR)
```

(Continued on next page)

68000 Cross Assembler (XASM)

(Listing continued, text begins on page 12)

Listing One

```
        OBJBUF(2)=OPNWRD(2)
        IF(OPNWC.EQ.2) OBJBUF(2)=OPNWRD(3)
        IF(OPNWC.EQ.2) OBJBUF(3)=OPNWRD(2)
        OBJWC=OBJWC+OPNWC
C
C....   NOW THAT WE HAVE IMMEDIATE DATA GET ,<EA>
C
        IF(OP2EA.EQ.0.AND.OP1EA.EQ.1) GOTO 6000
        IF(OP2EA.EQ.0) GOTO 615
C
C....   CHECK FOR  DATA,SR OR  DATA,CCR
C
        IF(OP2EA.LT.7) GOTO 612
        IF(OP2EA.GT.8) GOTO 8500
        IF((IMODE.EQ.1).AND.(OP2EA.EQ.8)) GOTO 611
        IF((IMODE.EQ.1).OR.(IMODE.EQ.3))  GOTO 8500
611     OBJBUF(1) = OPSKEL.OR.'74
        GOTO 6000
612     OBJBUF(1) = OPSKEL.OR.((OP2EA-1)*'10).OR.OP2DA
        GOTO 6000
C
C....   EVALUATE ,<EA> FOR COMPLEX ADR
C
615     CALL PROCOP(OPNPT2)
630     OBJBUF(OBJWC+1)=OPNWRD(2)
        IF(OPNWC.EQ.2) OBJBUF(OBJWC+1)=OPNWRD(3)
        IF(OPNWC.EQ.2) OBJBUF(OBJWC+2)=OPNWRD(2)
        OBJWC=OBJWC+OPNWC
        OBJBUF(1)=OBJBUF(1).OR.OPSKEL
        GOTO 6000
C
C....   EVALUATE DN,<EA>
C
620     OPSKEL=OPSKEL+(OP1DA*'1000).OR.'400
        IF(OP2EA.EQ.0) GOTO 615
        OBJBUF(1) = OPSKEL.OR.OP2DA.OR.((OP2EA-1)*'10)
        GOTO 6000
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS EOR INSTRUCTION
C       DN,<EA>   DATA,<EA>
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
700     IF(OP1EA.EQ.6) GOTO 610
        IF(OP1EA.NE.1) GOTO 8500
        IF(OP2EA.EQ.0) GOTO 620
        OBJBUF(1)=OPSKEL+((OP1EA-1)*'1000)+OP2DA+((OP1EA-1)*'10)
        GOTO 6000
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS ROTATES AND SHIFTS
C       DX,DY  DATA,DY  <EA>
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
800     IF(OP1EA.EQ.1.AND.OP2EA.EQ.1) GOTO 810
        IF(OP1EA.EQ.6.AND.OP2EA.EQ.1) GOTO 820
        IF(OP1EA.EQ.0.AND.OP2EA.EQ.1) GOTO 820
C
C....   PROCESS  <EA>
C
        IF(OP1EA.EQ.0) GOTO 801
        IF(OP1EA.LT.3.OR.OP1EA.GT.5) GOTO 8500
        OBJBUF(1)=OPSKEL+((OP1EA-1)*'10)+OP1DA
        GOTO 7000
C
801     CALL PROCOP(OPNPTR)
        OBJBUF(1)=OPSKEL+OPNWRD(1)
        OBJBUF(2) = OPNWRD(2)
        IF (OPNWC.EQ.2) OBJBUF(2) = OPNWRD(3)
        IF (OPNWC.EQ.2) OBJBUF(3) = OPNWRD(2)
        OBJWC = OBJWC + OPNWC
        GOTO 7000
C
810     OBJBUF(1) = OPSKEL.OR.'40.OR.(OP1DA*'1000).OR.OP2DA
        GOTO 6000
C
820     CALL PROCOP(OPNPTR)
        IF(OPNWRD(2).LT.1.OR.OPNWRD(2).GT.8) GOTO 8500
        IF(OPNWRD(2).EQ.8) OPNWRD(2)=0
        OBJBUF(1)=OPSKEL+(OPNWRD(2)*'1000)+OP2DA
        GOTO 6000
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS BRANCH INSTRUCTIONS
C       <LABEL>
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
900     IF(OPNPTR.EQ.0) GOTO 8500
        IF(OP1EA .NE.0) GOTO 8500
        BRFLG = 1
C
C....   GENERATE BRANCH ADDRESS
C
        CALL PROCOP(OPNPTR)
C
C....   CHK FOR FORCED SHORT ADR MODE
C
        IF(IMODE.EQ.4) GOTO 910
C
C....   CHECK FOR FWD REF SYMBOL OR REF BEFORE DEFINITION
C
        IF(OPNFLG.EQ.1) GOTO 905
C
C....   CHECK FOR SHORT BRANCH
C
        I = ICKVAL(OPNWRD(2))
        IF((I.EQ.0).AND.(OPNWRD(2).NE.'177600)) GOTO 910
        IF(IMODE.EQ.4) CALL ERROR(404)
C
C....   ELSE GENERATE TWO WORD BRANCH
C
905     OBJBUF(1) = OPSKEL
        OBJBUF(2) = OPNWRD(2)
        OBJWC = 2
        GOTO 920
C
C....   GENERATE SHORT BRANCH
C
910     OBJWC =1
        OPSKEL=OPSKEL+(OPNWRD(2).AND.'377)
        OBJBUF(1) = OPSKEL
920     BRFLG = 0
        GOTO 7000
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS BIT MODIFICATION INSTRUCTIONS
C       DN,<EA>   DATA,<EA>
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
1000    IF(OP1EA.EQ.1.OR.OP1EA.EQ.6) GOTO 1010
        GOTO 8500
1010    IF(OP1EA.EQ.6) GOTO 1020
        IF(OP2EA.EQ.0) GOTO 1015
C
C....   SIMPLE EA'S
C
        OBJBUF(1) = OPSKEL.OR.(OP1DA*'1000).OR.OP2DA
        OBJBUF(1) = OBJBUF(1) .OR. ((OP2EA-1)*'10)
        GOTO 7000
C
1015    CALL PROCOP(OPNPT2)
        OBJBUF(2) = OPNWRD(2)
        IF (OPNWC.EQ.2) OBJBUF(2) = OPNWRD(3)
        IF (OPNWC.EQ.2) OBJBUF(3) = OPNWRD(2)
        OBJWC = OBJWC + OPNWC
        OBJBUF(1) = OPSKEL.OR.OPNWRD(1).OR.(OP1DA*'1000)
        GOTO 7000
C
1020    CALL PROCOP(OPNPT2)
        IF(OPNWRD(3).NE.0) GOTO 8500
        OBJBUF(2)=OPNWRD(2)
```

(Continued on column 2)

(Continued on column 3)

(Continued next month.)

The Portable Pidgin

Z80 Macro-Assembly Implementation

by Herbert Gintis

Dr. Herbert Gintis, Dept. of Economics, Harvard University, Cambridge, MA 02138.

When in doubt, write in a machine-independent language. Anyone who has seen a cherished program lie fallow for lack of portability will recognize the wisdom of this maxim. Yet machine-independent languages are normally high-level affairs ill-suited for supporting the compact code and rapid execution times required of many systems programs. Whence my excitement upon learning of William Gale's *Pidgin*, a low-level, structured, portable programming language (*DDJ* #57). I suspect that Dr. Gale's compiler generator Meta4 (*DDJ* #58) is only the first of many powerful systems programs that will be written in the flexible format of Pidgin.

Pidgin is portable because it was designed to be supported by a portable compiler. Indeed, Dr. Gale has provided us with Tincmp, a Pidgin compiler written in Pidgin itself, yet organized to facilitate its relatively easy implementation on any machine. But only relatively! What I shall suggest here is that Tincmp can be implemented with little difficulty on any machine for which a reasonably powerful macro-assembler exists. The example presented here uses a Z80 microprocessor running under CP/M, and the macroprocessor is Venus, a structured editor-assembler-linker-symbolic debugger which I've developed over the past year.

The strategy for implementing Tincmp is straightforward: generate an assembly-language version of the compiler by translating each and every Pidgin statement into a macro.

The result of this exercise is a new Tincmp source file which I have called Tinsource, shown in Listing 2, and a source file of macros called Tinmacro, shown in Listing 1. I shall discuss these two source files in turn.

Since Venus uses standard assembly-language pseudo-ops (e.g., CP/M's ASM.-COM), Tinsource should be easy to understand, subject to the following notes. First, Venus uses the pseudo-op INCBG <filename.filetype> to instruct the assembler to include the specified file at the beginning of the source code before assembly. In this case, Tinmacro will be so included. Second, a string of one to five alphanumerics beginning with a letter, occurring where Venus expects a mnemonic opcode or pseudo-op, and followed by an exclamation point (!), is treated as a macro call.

The meaning of the macros is either the same as that of the corresponding Pidgin statement, is indicated in the remarks, or can be inferred by comparing Tinsource with Tincmp. I have attempted to use obvious mnemonics where possible. Thus "B" stands for "byte", "I" for "int", "E" for "set equal to", "O" for "or", "A" for "and", "EE" for "is equal to", "LE" for "is less than or equal to", and so forth.

Since Venus is a structured assembler, I in fact implemented the control statements WHILE-ON-ENDWHILE and IF-ELSE-ENDIF directly. However, since structured facilities are rare in assemblers, I have rewritten Tinsource by treating these control statements as program labels. Thus a WHILE becomes the string "WH" followed by a number, and ENDWHILE becomes "NDWH" followed by a number. The ON in the WHILE-ENDWHILE construction is translated into the macro IFNOT, which moves beyond the ENDWHILE if the condition tested is false. The IF-ELSE-ENDIF is similarly treated, using the common beginning string IF followed by a number. The CHOOSE ON-CASE construction is directly implemented as a macro.

Tinmacro, given in Listing 1 is also by and large self-explanatory. The macroprocessor of Venus is fairly standard. The name of the macro is written in the label column followed by the pseudo-op MACRO, followed in turn by a list of the prototype variables separated by commas. A proto-type variable is preceded by the ampersand (&). The period (.) is a non-printed string terminator. Thus if the macroprocessor encounters the line

```
LD HL, (I.&DEST)
```

and the current value of &DEST is GG, the line will be expanded as

```
LD HL, (IGG)
```

Tinmacro is organized according to the principle that each byte variable in the object code generated (which is in this case an assembly source file) is preceded by the string BB, and each integer variable is preceded by the single letter I. By comparing the listing of Tinmacro with the 8080A macros suggested by Andrew Bender (*DDJ* #65 – March 1982), the reader will notice that I have streamlined some of the coding, using the fact that the Z80 accumulator can be loaded without altering the flag conditions.

To implement Tincmp for your machine, alter Tinmacro and Tinsource to meet the conventions of your assembler. If you do not have a pseudo-op like INCBG, then Tinmacro will have to be merged directly with Tinsource, an easy task on a text editor. If you do not have a macroprocessor, you are probably out of luck, since the full source code of Tinsource with all macros expanded would probably run some 4000 lines.

When Tinsource assembles without errors, then you must implement the Tincmp macros, following the example of Bender's, perhaps with the streamlining suggested in Listing 1. Note that Bender's implementation of the structured control statements uses multiple ORG's, which are disliked by some relocating assemblers. Venus does not mind such multiple ORG's, but there is an easy alternative strategy which I have successfully tested anyway.

The problem arises with the CHOOSE ON $$ and CASE $$ statements in Pidgin. Unlike the C language, Pidgin does not allow a program to "fall through" to the next CASE upon the execution of one CASE, but rather demands exiting the control structure. Thus in generating the assembly source code for the CASE $$ statement, the first line must be a jump over any remaining CASE $$ statements. However, this jump must be suppressed in the *first* such statement after the CHOOSE ON $$. The strategy for handling this is to have the CHOOSE ON $$ place a comment symbol (;) on the top of the stack, and have each CASE $$ immediately place the top of the stack in the destination file, and replace it with a blank ( ), which will be ignored by the assembler. The relevant Tincmp macros will then be:

```
:CHOOSE ON $$;
LD A, (BB↑P1C↑P2C) ;
LD HL,ERASER;
LD (HL),A↑U0S↑L@;S↑U1S;
```

```
:CASE $$;
↑!0P↑!9P↑!8P;
↑P9CJP NDCH↑P0N;
CS↑P8N LD A, (BB↑P1C↑P2C) ;
CP (HL) ;
JP NZ,CS↑U1S↑S8N↑L@ S↑P0S;
```

The final step in implementing Tincmp involves developing the system utilities, including the algebraic routines ICOMP, ISUB, CDEHL, IMUL, and IDIV, as well as the CP/M input-output routines START, BREAD, BWRITE, PRT, IOPEN, MSG, CONOUT, and BCLOSE. The requirements for the algebraic routines are well-described in Bender's excellent article. CONOUT is essentially CP/M's Direct Consol I/O (Function 6), but preserving all registers. PRT uses CONOUT to output the string pointed to by the HL register; the string must terminate with a carriage return (0dh in ASCII). MSG is the same as PRT, except that precisely 9 characters are transmitted. The I/O routines must be written using the system utilities associated with your assembler. Their functions will be obvious from the places in Listing 1 where they are used, after a careful reading of the discussion of these routines in Bender's article.

DDJ

(Listing begins at right)

Tinmacro (Text begins on page 25)

Listing One

```
(A:TINMACRO)li 11,f
1 *
2 * *****          TINMACRO           *****
3 *
4 ****************************************
5 *                                      *
6 *               Listing 1              *
7 *                                      *
8 ****************************************

9 *
10 * Macros for the implementation of TINSOURCE
11 *
12 IIEI          MACRO   &A,&B,&C  ;I$$(I$$)=I$$
13               LD      HL,(I.&B)
14               LD      DE,I.&A
15               ADD     HL,HL
16               ADD     HL,DE
17               EX      DE.HL
18               LD      HL,(I.&C)
19               LD      A,L
20               LD      (DE),A
21               INC     DE
22               LD      A,H
23               LD      (DE),A
24               MEND
25 IINC          MACRO   &A          ;I$$++
26               LD      HL,(I.&A)
27               INC     HL
28               LD      (I.&A),HL
29               MEND
30 BIEB          MACRO   &A,&B,&C  ;$$(I$$)=$$
31               LD      HL,(I.&B)
32               LD      DE.BB.&A
33               ADD     HL,DE
34               LD      A,(BB.&C)
35               LD      (HL),A
36               MEND
37 BINC          MACRO   &A          ;$$++
38               LD      HL,BB.&A
39               INC     (HL)
40               MEND
41 BBEB          MACRO   &A,&B,&C  ; $$($$)=$$
42               LD      HL,(BB.&B)
43               LD      H,0
44               LD      DE,BB.&A
45               ADD     HL,DE
46               LD      A,(BB.&C)
47               LD      (HL),A
48               MEND
49 BEBB          MACRO   &A,&B,&C  ;$$=$$($$)
50               LD      HL,(BB.&C)
51               LD      H,0
52               LD      DE,BB.&B
53               ADD     HL,DE
54               LD      A,(HL)
55               LD      (BB.&A),A
56               MEND
57 IEIPI         MACRO   &A,&B,&C  ;I$$=I$$+I$$
58               LD      HL,(I.&B)
```

(Continued on column 2)

```
59               LD      DE,(I.&C)
60               ADD     HL,DE
61               LD      (I.&A),HL
62               MEND
63 IEII          MACRO   &A,&B,&C  ;I$$=I$$(I$$)
64               LD      HL,(I.&C)
65               LD      DE,I.&B
66               ADD     HL,HL
67               ADD     HL,DE
68               LD      E,(HL)
69               INC     HL
70               LD      D,(HL)
71               EX      DE,HL
72               LD      (I.&A),HL
73               MEND
74 BEBI          MACRO   &A,&B,&C  ;$$=$$(I$$)
75               LD      HL,(I.&C)
76               LD      DE,BB.&B
77               ADD     HL,DE
78               LD      A,(HL)
79               LD      (BB.&A),A
80               MEND
81 BEB           MACRO   &A,&B       ;$$=$$
82               LD      A,(BB.&B)
83               LD      (BB.&A),A
84               MEND
85 IEI           MACRO   &A,&B       ;I$$=I$$
86               LD      HL,(I.&B)
87               LD      (I.&A),HL
88               MEND
89 BECON         MACRO   &A,&B       ;$$=+$$$
90               LD      A,&B
91               LD      (BB.&A),A
92               MEND
93 IECON         MACRO   &A,&B       ;I$$=+$$$$$
94               LD      HL,&B
95               LD      (I.&A),HL
96               MEND
97 BEBPB         MACRO   &A,&B,&C  ;$$=$$+$$
98               LD      HL,BB.&C
99               LD      A,(BB.&B)
100               ADD     A,(HL)
101               LD      (BB.&A),A
102               MEND
103 BEBMB         MACRO   &A,&B,&C  ;$$=$$-$$
104               LD      HL,BB.&C
105               LD      A,(BB.&B)
106               SUB     (HL)
107               LD      (BB.&A),A
108               MEND
109 BEBAB         MACRO   &A,&B,&C  ;$$=$$&$$
110               LD      HL,BB.&C
111               LD      A,(BB.&B)
112               AND     (HL)
113               LD      (BB.&A),A
114               MEND
115 BEBOB         MACRO   &A,&B,&C  ;$$=$$?$$
116               LD      HL,BB.&C
117               LD      A,(BB.&B)
118               OR      (HL)
119               LD      (BB.&A),A
120               MEND
121 IEMI          MACRO   &A,&B       ;I$$=-I$$
122               LD      HL,(I.&B)
123               CALL    ICOMP
124               LD      (I.&A),HL
```

(Continued on next page)

Tinmacro (Listing continued, text begins on page 25)

Listing One *(Continued)*

```
125 ICOMP       EXTRN
126             MEND
127 IEIMI       MACRO   &A,&B,&C   ;I$$=I$$-I$$
128             LD      HL,(I.&C)
129             EX      DE,HL
130             LD      HL,(I.&B)
131             CALL    ISUB
132             LD      (I.&A),HL
133 ISUB        EXTRN
134             MEND
135 INEI        MACRO   &BB,&IAA,&IBB   ;$$=I$$!=I$$
136             LD      DE,I.&IBB
137             LD      HL,I.&IAA
138             CALL    INTCK
139             LD      (BB.&BB),A
140 INTCK       EXTRN
141             MEND
142 ILEI        MACRO   &BB,&IAA,&IBB   ;$$=I$$(=I$$
143             LD      HL,(I.&IBB)
144             EX      DE,HL
145             LD      HL,(I.&IAA)
146             CALL    CDEHL
147 CDEHL       EXTRN
148             CALL    SETAC
149 SETAC       EXTRN
150             LD      (BB.&BB),A
151             MEND
152 BYTE        MACRO   &BB         ;BYTE $$
153             RAM
154 BB.&BB      DS      1
155             REL
156             MEND
157 INT         MACRO   &IAA        ;INT I$$
158             RAM
159 I.&IAA      DS      2
160             REL
161             MEND
162 BYTEB       MACRO   &B1,&NNN    ;BYTE $$($$$) OR BYTE $$($$$$$)
163             RAM
164 BB.&B1      DS      &NNN
165             REL
166             MEND
167 INTI        MACRO   &IAA,&NNN   ;INT I$$($$$$$)
168             RAM
169 I.&IAA      DS      2 * &NNN
170             REL
171             MEND
172 GOSUB       MACRO   &ROUTINE
173             CALL    SUB.&ROUTINE
174             MEND
175 STOP        MACRO   &N
176             LD      HL,$+6
177             CALL    PRT
178             JP      0000
179             DB      'STOP ON ',&N+'0',:D
180             MEND
181 ISLI        MACRO   &BB,&IAA,&IBB   ;$$=I$$(!I$$
182             LD      HL,(I.&IBB)
183             EX      DE,HL
184             LD      HL,(I.&IAA)
185             CALL    CDEHL
186 CDEHL       EXTRN
187             RLA
188             CP      (HL)
189             RLA
190             AND     1
191             LD      (BB.&B1),A
192             MEND
193 IEIB        MACRO   &IAA,&IBB,&B1   ;I$$=I$$($$)
194             LD      HL,(BB.&B1)
195             LD      H,0
196             LD      DE,I.&IBB
197             ADD     HL,HL
198             ADD     HL,DE
199             LD      DE,I.&IAA
200             LD      A,(HL)
201             LD      (DE),A
202             INC     HL
203             INC     DE
204             LD      A,(HL)
205             LD      (DE),A
206             MEND
207 BEI         MACRO   &B1,&IAA    ;$$=I$$
208             LD      A,(I.&IAA)
209             LD      (BB.&B1),A
210             MEND
211 BLEB        MACRO   &B1,&B2,&B3   ;$$=$$(=$$
212             LD      HL,BB.&B3
213             LD      A,(BB.&B2)
214             CP      (HL)
215             CALL    SETAC
216             LD      (BB.&B1),A
217 SETAC       EXTRN
218             MEND
219 BDEC        MACRO   &B1         ;$$--
220             LD      HL,BB.&B1
221             DEC     (HL)
222             MEND
223 ITMS        MACRO   &IAA,&IBB,&ICC   ;I$$=I$$*I$$
224             LD      HL,(I.&ICC)
225             EX      DE,HL
226             LD      HL,(I.&IBB)
227             CALL    IMUL
228             LD      (I.&IAA),HL
229 IMUL        EXTRN
230             MEND
231 IDV         MACRO   &IAA,&IBB,&ICC   ;I$$=I$$/I$$
232             LD      HL,(I.&IBB)
233             EX      DE,HL
234             LD      HL,(I.&ICC)
235             CALL    IDIV
236             LD      (I.&IAA),HL
237 IDIV        EXTRN
238             MEND
239 IBEI        MACRO   &IAA,&B1,&ICC   ;I$$($$)=I$$
240             LD      HL,(BB.&B1)
241             LD      H,0
242             ADD     HL,HL
243             LD      DE,I.&IAA
244             ADD     HL,DE
245             LD      DE,I.&ICC
246             LD      A,(DE)
247             LD      (HL),A
248             INC     DE
249             INC     HL
250             LD      A,(DE)
251             LD      (HL),A
252             MEND
253 ITBB        MACRO   &IAA,&B1,&B2   ;UNPACK(I$$,$$,$$)
254             LD      HL,(I.&IAA)
255             LD      A,L
256             LD      (BB.&B1),A
```

```
188                AND     1
189                LD      (BB.&BB),A
190                MEND
191 RDFIL          MACRO   &CC,&F1    ;READ $$ FROM $$
192                LD      HL,BB.&CC
193                LD      DE,BB.&F1
194                CALL    BREAD
195 BREAD          EXTRN
196                MEND
197 WRFIL          MACRO   &CC,&F1    ;WRITE $$ INTO $$
198                LD      A,(BB.&CC)
199                LD      DE,BB.&F1
200                CALL    BWRITE
201                LD      A,(ER)
202                OR      A
203                CALL    NZ,PRT
204 BWRITE         EXTRN
205 PRT            EXTRN
206                MEND
207 OPEN           MACRO   &F1,&RW,&INM  ;OPEN $$ FOR $$ AT I$$
208                LD      DE,BB.&F1
209                LD      A,(BB.&RW)
210                LD      HL,(I.&INM)
211                CALL    IOPEN
212                LD      A,(ER)
213                OR      A
214                CALL    NZ,PRT
215 PRT            EXTRN
216 ER             EXTRN
217 IOPEN          EXTRN
218                MEND
219 MS             MACRO   &STRING    ;PRINT 9 CHARS
220                LD      HL,$+5
221                CALL    MSG
222 MSG            EXTRN
223                JR      MS.&XX.FF
224                DB      &STRING
225 MS.&XX.FF      REM
226                MEND
227 BNEB           MACRO   &B1,&B2.&B3   ;$$=$$'=$$
228                LD      HL,BB.&B3
229                LD      A,(BB.&B2)
230                CP      (HL)
231                LD      A,0
232                JR      Z,1
233                INC     A
234                LD      (BB.&B1),A
235                MEND
236 BEEB           MACRO   &B1,&B2,&B3   ;$$=$$==$$
237                LD      HL,BB.&B3
238                LD      A,(BB.&B2)
239                CP      (HL)
240                LD      A,1
241                JR      Z,1
242                XOR     A
243                LD      (BB.&B1),A
244                MEND
245 IEB            MACRO   &IAA,&B1   ;I$$=$$
246                LD      A,(BB.&B1)
247                LD      L,A
248                LD      H,0
249                LD      (I.&IAA),HL
250                MEND
251 BSLB           MACRO   &B1,&B2,&B3   ;$$=$$('$$
252                LD      A,(BB.&B2)
253                LD      HL,BB.&B3
```

(Continued on top of page 28)

```
323                LD      A,H
324                LD      (BB.&B2),A
325                MEND
326 IEEI           MACRO   &B1,&IAA,&IBB   ;$$=I$$==I$$
327                LD      HL,(I.&IAA)
328                LD      DE,(I.&IBB)
329                CALL    CDEHL
330                LD      A,1
331                JR      Z,1
332                XOR     A
333                LD      (BB.&B1),A
334 CDEHL          EXTRN
335                MEND
336 GOTO           MACRO   &NN         ;GOTO $$
337                JP      LOC.&NN
338                MEND
339 WRITE          MACRO   &B1         ;WRITE $$
340                LD      A,(BB.&B1)
341                CALL    CONOUT
342 CONOUT         EXTRN
343                MEND
344 CLOSE          MACRO   &B1         ;CLOSE $$
345                LD      HL,BB.&B1
346                CALL    BCLOSE
347                LD      A,(ER)
348                OR      A
349                CALL    NZ,PRT
350 BCLOSE         EXTRN
351                MEND
352 CHSON          MACRO   &B1         ;CHOOSE ON $$
353                LD      A,(BB.&B1)
354                MEND
355 IFNOT          MACRO   &B1,&DEST   ;GOTO DEST IF B1 FALSE
356                LD      A,(BB.&B1)
357                OR      A
358                JP      Z,&DEST
359                MEND
360 CASE           MACRO   &B1,&DEST   ; GOTO DEST IF [A] != [&B1]
361                LD      HL,BB.&B1
362                CP      (HL)
363                JP      NZ,&DEST
364                MEND
(A:TINMACRO)ff
```

Tinsource

Listing Two

```
(A:TINMACRO)b
(A:TINCMP)la 11.f
1 *
2 * *****          TINSOURCE          *****
3 *
4 ****************************************
5 *                                      *
6 *                LISTING 2             *
7 *                                      *
8 ****************************************
9 *
10 * The TINCMP Compiler in Assembly Language
11 *
12                UNLST               ; suppress printer listing
13                INCBG  TINMACRO     ; append TINMACRO here
14                LIST                ; resume printer listing
```

(Continued on next page)

Tinsource (Listing continued, text begins on page 25)

Listing Two *(Continued)*

```
                XMAC                ; suppress the printing of macro expansion
ER              EXTRN
BBER            EQU     ER          ; all byte variables begin with BB
                BYTE!   AA          ;WORK
                BYTE!   BB          ;WORK
                BYTE!   DD          ;WORK
                BYTE!   EE          ;WORK
                BYTEB!  BF,080      ;EXPANSION BUFFER
                BYTE!   BL          ;BLANK
                BYTE!   BP          ;PTR INTO BF
                BYTE!   C0          ;0
                BYTE!   C1          ;1
                BYTE!   C2          ;2
                BYTE!   C3          ;3
                BYTE!   C4          ;40
                BYTE!   C8          ;80
                BYTE!   C9          ;9
                BYTE!   CC          ;INPUT CHAR
                BYTE!   CX          ;10
*
                BYTE!   DG          ;DIGIT FROM PARAMETER DEF
                BYTEB!  DS,10       ;DIGIT STACK FOR SUB SD
                BYTE!   EF          ;END OF FILE MARK
                BYTEB!  F1,128      ;INPUT BUFFER
                BYTEB!  F2,128      ;OUTPUT BUFFER
                BYTE!   HA          ;'A'
                BYTE!   HF          ;'F'
                BYTE!   LE          ;END OF LIST
                BYTE!   LF          ;LINE FEED CHAR
                BYTEB!  LS,9000     ;LIST OF MACRO DEFS
                BYTE!   MF          ;MACRO REPLACEMENT OPERATOR FLAG
                BYTE!   ML          ;MACRO LENGTH
                BYTE!   MM          ;MIN. MACRO LENGTH
*
                BYTE!   ND          ;# OF DIGITS IN SD OUTPUT
                BYTE!   NL          ;NEW LINE
                BYTE!   O1          ;FETCH CODE
                BYTE!   O2          ;INDEX CODE
                BYTE!   O3          ;DISPOSE CODE
                BYTE!   OA          ;'+' OPERAOR
                BYTE!   OB          ;'!' OPERATOR
                BYTE!   OC          ;'C' CHAR. DISPOSE OPERATOR
                BYTE!   OD          ;'V' DIGIT CONV. FETCH
                BYTE!   OE          ;ESC CHAR
                BYTE!   OG          ;IGNORE CHAR
                BYTE!   OH          ;'H' HEX CONV. FETCH
                BYTE!   OL          ;'L' LITERAL FETCH
                BYTE!   OM          ;'BYTE! ' MULT DISPOSE
                BYTE!   ON          ;'N' NUMERIC LITERAL FETCH
*
                BYTE!   OP          ;'P' PARAMETER FETCH OR DISPOSE
                BYTE!   OR          ;'-' REDUCE (SUBTRACT) DISPOSE
                BYTE!   OS          ;'S' STACK FETCH OR DISPOSE
*
                BYTE!   OT          ;TRACE FLAG
                BYTE!   PP          ;PTR INTO IPR
                BYTE!   RB          ;BEGIN DEF. FLAG
                BYTE!   RC          ;END OF LINE (COMMENT) FLAG
                BYTE!   SF          ;SUBS PARAMETER FLAG
                BYTE!   SP          ;STACK PTR
                BYTE!   TR          ;TRUE IF NO TRACE
                BYTE!   UG          ;USE IGNORE; TRUE UNLESS OG='X'
                BYTE!   UN          ;FLAG FOR NOT SUPRESSING NEW LINESON OUTPUT
                BYTE!   UO          ;USE OPERATIONS; TRUE UNLESS MF='X'

                BEB!    CC,NL
IF32
IF33
                BNEB!   AA,CC,NL    ;AA=CC!=NL
                BNEB!   BB,BP,C8
                BEBAB!  AA,AA,BB
                IFNOT!  AA,NDWH9    ;WHILE ( 80 CHARS AND NOT NL
                BBEB!   BF,BP,CC    ;PUT IN BUFFER
                BINC!   BP
                JP      WH9
NDWH9
WH10            BNEB!   AA,CC,NL
                IFNOT!  AA,NDWH10   ;DUMP LONG LINE INPUT
                GOSUB!  GC          ;DUMP LONG LINE INPLT
                JR      WH10
NDWH10
                BBEB!   BF,BP,RC    ;EOL FLAG
                BINC!   BP
                BBEB!   BF,BP,NL
                BEB!    LE,BP       ;END OF LIST
                BLEB!   AA,BP,MM    ;AA=BP(=MM
                IFNOT!  AA,IF34     ;TOO SHORT TO MATCH
                BECON!  ML,0
                GOTO!   17
IF34            BECON!  ML,1
                IEI!    IDP,I00     ;IDP=I00
                BEB!    PP,C0
                IEI!    IJJ,I00
                IEB!    INM,C0      ;INM=C0
WH11            ISLI!   AA,IDP,IED  ;AA=IDP(!IED ;END OF DEFS
                IFNOT!  AA,NDWH11
                BEB!    BP,C0
WH12            BLEB!   AA,BP,LE
                IFNOT!  AA,NDWH12
                BEBI!   AA,LS,IJJ   ;AA=LS(IJJ)
                BEEB!   AA,AA,RC    ;AA=AA==RC
                BEBB!   O3,BF,BP    ;O3=BF(BP)
                BEEB!   O3,O3,RC
                BEBAB!  AA,AA,O3    ;AA=AA&O3
                IFNOT!  AA,IF35
                GOSUB!  DM          ;DO MACRO EXPANSION
                GOTO!   00
IF35            BEBB!   AA,BF,BP
                BEBI!   BB,LS,IJJ
                BEEB!   AA,AA,BB
                IFNOT!  AA.IF36
                GOTO!   01          ;MATCHING
IF36            BNEB!   AA,BB,SF    ;NOT TEMPLATE PARAMETER FLAG
                IFNOT!  AA,IF37
                GOTO!   10          ;MISMATCHED
IF37            BINC!   PP THIS IS PARAMETER
                BEBB!   AA,BF,BP
                IEB!    IAA,AA
                IBEI!   IPR,PP,IAA  ;IPR(PP)=IAA
LOC01           BINC!   BP
                IINC!   IJJ
                JP      WH12
NDWH12
LOC10           BEB!    PP,C0
                IINC!   INM
                IEII!   IDP,ILP,INM  ;IDP=ILP(INM)
                IEI!    IJJ,IDP
                JP      WH11
NDWH11
LOC17           BEB!    BP,C0
WH13            BEBB!   CC,BF,BP    ;CC=BF(BP)
                BEBPB!  O1,BP,C1    ;O1=BP+C1
                BEBB!   AA,BF,O1    ;AA=BF(O1)
                BNEB!   AA,AA,NL
```

```
79                BYTE!  UT          ;USE TRACE
80                BYTE!  ZR          ;'0'
81                INT!   I00         ;0
82                INT!   I01         ;1
83                INT!   I09         ;9
84                INT!   I10         ;10
85                INT!   I16         ;16
86                INT!   IAA         ;WORK
87                INT!   IBB         ;WORK
88                INT!   IBC         ;FILE NUMBER
89                INT!   IDP         ;DEF PTR WHILE MATCH
90                INT!   IED         ;PTR TO END OF DEFS
91                INT!   III         ;PTR TO L WHILE READING
92                INT!   IJJ         ;PTR TO L
93                INT!   ILM         ;MAX LIMIT FOR STORING IN L
94                INTI!  ILP,1000    ;PTRS TO MACROS
95                INT!   IMP         ;MACRO PTR DURING EXPANSIONS
96                INT!   INM         ;# OF MACROS
97                INTI!  IPR,10      ;PARAMETER VALUES
98                INTI!  ISS,40      ;MAIN STACK
99                INT!   ITU         ;VALUE OF PARAMETER TO USE
100                INT!   IUU         ;SYMBOL GENERATOR
101                INT!   IXX         ;WORK
102                INT!   IYY         ;WORK
103 * BEGINMAIN(AC,IAV)
104               CALL   START
105 START         EXTRN
106               BECON! NL,:D
107               BECON! LF,:A
108               GOSUB! CR
109               MS!    'Copyright'
110               MS!    ' (C) 1982'
111               MS!    ' Herbert '
112               MS!    'Gintis   '
113               GOSUB! CR
114               MS!    'Z80 TINCM'
115               MS!    'P Compile'
116               MS!    'r CPM Ver'
117               MS!    'sion 1.1 '
118               GOSUB! CR
119               GOSUB! IN          ;INITIALIZE
120               GOSUB! RM          ;READ MACRO'S
121 LOC00
122 WH7           GOSUB! GI          ;READ CC FROM F1
123               BEEB!  AA,ER,C0
124               IFNOT! AA,NDWH7
125               IFNOT! UG,IF31
126 WH8           BEEB!  BB,CC,NL    ;BB=CC==NL
127               BEEB!  DD,CC,LF
128               BEEB!  AA,CC,OG
129               BEBOB! EE,BB,DD    ;EE=BB?DD
130               BEEB!  BB,CC,BL
131               BEBOB! AA,BB,AA
132               BEBOB! AA,EE,AA
133               IFNOT! AA,NDWH8    ;IGNORE LEADING CHARS (NL,LF,OG,BL)
134               GOSUB! GC
135               JR     WH8
136 NDWH8
137 IF31
138               BEB!   BP,C1       ;BUF PTR
139               BBEB!  BF,C0,CC    ;BF(C0)=CC
140 WH9           GOSUB! GC
141               BEEB!  AA,CC,NL
142               IFNOT! AA,IF32
143               GOSUB! GC
144               BEEB!  AA,CC,LF
145               IFNOT! AA,IF33
```

(Continued on top of page 30)

```
215               IFNOT! AA,NDWH13
216               IFNOT! ML,IF38
217               WRITE! CC
218 IF38          BEB!   AA,CC
219               GOSUB! WA          ;WRITE CC INTO F2
220               BINC!  BP
221               JP     WH13
222 NDWH13
223               IFNOT! ML,IF39
224               GOSUB! CR
225 IF39          IFNOT! UN,IF40
226               BEB!   AA,NL
227               GOSUB! WA
228               BEB!   AA,LF
229               GOSUB! WA
230 IF40
231               JP     WH7
232 NDWH7
233 LOC88         MS!    'TINCMP CO'
234               MS!    'MPILATION'
235               MS!    ' FINISHED'
236               GOSUB! CR
237               CLOSE! F1
238               CLOSE! F2
239               JP     :100
240 *
241 * Initialize
242 *
243 SUBIN         IECON! ILM,8920    ;ILM=+08920
244               IECON! I00,0
245               IECON! I01,1
246               IECON! I10,10
247               IECON! I09,9
248               BECON! C0,0        ; CO=+000
249               BECON! C1,1
250               BECON! C2,2
251               BECON! C3,3
252               BECON! EF,:1A
253               BECON! C4,40
254               BECON! C8,80
255               IECON! I16,16
256               BECON! SP,0
257               BECON! C9,'9'
258               BECON! ZR,'0'
259               BECON! BL,' '
260               BECON! HF,'F'
261               BECON! HA,'A'
262               BECON! CX,10
263               IEI!   IBC,I01     ;IBC=I01
264               BECON! TR,'R'
265 * CLOSE F1
266 * ASSOCIATE FCB 1 WITH IBC
267               OPEN!  F1,TR,IBC   ;OPEN F1 FOR TR AT IBC
268               BECON! TR,'W'      ;TR='W'
269               IINC!  IBC         ;IBC++
270 * CLOSE F2
271 * ASSOCIATE FCB 2 WITH IBC
272               OPEN!  F2,TR,IBC
273               GOSUB! GI          ;READ AA FROM F1;'X' SUPRESSES NEW LINE OUTPUT
274               BEB!   AA,CC
275               BECON! OT,'T'
276               BECON! UT,0
277               BECON! BB,'X'
278               BNEB!  UN,AA,BB    ;UN=AA!=BB;TRUE IF NOT SUPRESS
279               GOSUB! GI          ;BEGIN DEF FLAG
280               BEB!   RB,CC
281               GOSUB! GI          ;COMMEND & EOL FLAG
```

(Continued on next page)

Tinsource (Listing continued, text begins on page 25)

Listing Two *(Continued)*

```
282                BEB!   RC,CC
283                GOSUB! GI          ;TEMPLATE PARAMETER FLAG
284                BEB!   SF,CC
285                GOSUB! GI          ;EXPANSION OP FLAG
286                BEB!   MF,CC
287 *
288 * Set UO true iff expansion flag is not 'X'
289 *
290                BECON! BB,'X'
291                BEEB!  AA,MF,BB    ;AA=MF==BB
292                IFNOT! AA,IF1
293                BEB!   UO,C0
294                JR     IF2
295 IF1            BEB!   UO,C1
296 IF2
297                BECON! OP,'P'      ;PARAMETER DESIGNATOR IN OPERATION SEQ
298                BECON! OE,'@'      ;ESC CHAR
299                BECON! OD,'V'      ;CONVERT PARAM
300                BECON! OB,''       ;POP STACK DESIG
301                BECON! OS,'S'      ;STACK DESIG
302                BECON! OH,'H'      ;HEX CONST FETCH &WRITE
303                BECON! ON,'N'      ;LITERAL NUMERIC FETCH
304                BECON! OL,'L'      ;LITERAL BYTE FETCH
305                BECON! OC,'C'      ;CHAR OUT DESIG
306                BECON! OA,'+'      ;ADD TO STACK DESIG
307                BECON! OR,'-'      ;SUB FROM STACK
308                BECON! OM,'*'      ;MULTIPLY STACK BY BASE & ADD
309                GOSUB! GI          ;IGNORE CHAR
310                BEB!   OG,CC
311                BECON! AA,'X'      ;AA='X'
312                BEEB!  BB,AA,OG    ;BB=AA==OG
313                IFNOT! BB,IF3
314                BECON! UG,0
315                JR     IF4
316 IF3            BECON! UG,1
317 IF4
318                GOSUB! GI          ;NEW LINE
319                BNEB!  AA,NL,CC    ;AA=NL'=CC
320                IFNOT! AA,IF5
321                MS!    'FLAG LINE'
322                STOP!  1
323 IF5            IECON! IUU,100
324                RET                ;ENDSUB IN
325 *
326 * Read Macros
327 *
328 SUBRM         IEI!   III,I00
329                IEB!   INM,C0      ;INM=C0
330                BECON! MM,127      ;MM=127
331 WH1
332                GOSUB! GI
333                BEEB!  AA,ER,C0    ;AA=ER==C0
334                IFNOT! AA,NDWH1
335                CHSON' CC          ;CHOOSE ON CC
336                CASE!  OE,CS1      ;ESC CHAR
337                GOSUB! GI
338                JP     LOC77
339 CS1            CASE!  RB,CS2      ;MAC DEF FLAG
340                IIEI!  ILP,INM,III   ;ILP(INM)=III
341                IINC!  INM
342                BECON! ML,0
343                JP     NDCH1
344 CS2            CASE!  NL,CS2A     ;IGNORE NL
345                JP     NDCH1
```

```
412                WRITE! LF
413                RET                ;ENDSUB CR
414 *
415 * Do Macro Expansion
416 *
417 SUBDM         IEIPI! IMP,IJJ,I01   ;IMP=IJJ+I01
418                IINC!  INM
419                IEII!  IDP,ILP,INM   ;IDP=ILP(INM)
420 WH3            ISLI!  AA,IMP,IDP   ;AA=IMP('IDP
421                IFNOT! AA,NDWH3
422                BEBI!  AA,LS,IMP   ;AA=LS(IMP)
423                IFNOT! UO,IF11
424                BEEB!  AA,AA,MF    ;AA=AA==MF
425                JR     IF12
426 IF11           BEB!   AA,C0
427 IF12
428                IFNOT! AA,IF13
429                IINC!  IMP
430                BEBI!  01,LS,IMP   ;01=LS(IMP)
431                IINC!  IMP
432                BEBI!  AA,LS,IMP   ;AA=LS(IMP)
433                BEB!   02,AA
434                GOSUB! CD          ;FOR DIGIT CONVERSION
435                BEB!   DG,AA
436                IINC!  IMP
437                BEBI!  03,LS,IMP
438                IFNOT! UT,IF15
439                WRITE! 01
440                WRITE! 02
441                WRITE! 03
442 IF15
443                CHSON! 01          ;CHOOSE ON 01
444                CASE!  OP,CS4A     ;FETCH PARAMETER
445                IEIB!  ITU,IPR,DG   ;ITU=IPR(DG)
446                JP     NDCH2
447 CS4A           CASE!  OD,CS5      ;CONVERT FROM DIGIT TO CHAR
448                IEIB!  IAA,IPR,DG   ;IAA=IPR(DG)
449                BEI!   AA,IAA      ;AA=IAA
450                GOSUB! CD
451                IEB!   ITU,AA      ;ITU=AA
452                JP     NDCH2
453 CS5            CASE!  OB,CS6      ;POP STACK
454                IEIB!  ITU,ISS,SP   ;ITU=ISS(SP)
455                BLEB!  AA,SP,C0    ;AA=SP(=C0
456                IFNOT! AA,IF16
457                MS!    'S STACKER'
458                GOSUB! CR
459                BEB!   SP,C1
460 IF16
461                BDEC!  SP
462                JP     NDCH2
463 CS6            CASE!  OS,CS7      ;FETCH TOP STACK, NO POP
464                IEIB!  ITU,ISS,SP   ;ITU=ISS(SP)
465                JP     NDCH2
466 CS7            CASE!  OH,CS8      ;FETCH & WRITE HEX CONST. BYTE
467                BEB!   AA,02
468                GOSUB! CH
469                IEB!   IAA,AA      ;IAA=AA
470                ITMS!  IAA,IAA,I16 IAA=IAA*I16
471                BEB!   AA,03
472                GOSUB! CH
473                IEB!   IBB,AA
474                IEIPI! ITU,IAA,IBB
475                BEB!   03,0C
476                JP     NDCH2
477 CS8            CASE!  OL,CS9      ;LITERAL BYTE FETCH
478                IEB!   ITU,02      ;ITU=02
479                JP     NDCH2
480 CS9            CASE!  ON,CS10     ;LITERAL DIGIT FETCH
```

```
346 CS2A        CASE!  LF,CS2B    ;IGNORE LF
347             JP     NDCH1
348 CS2B        CASE!  RC,CS3     ;IGNORE COMMENTS
349             BIEB!  LS,III,RC  ;LS(III)=RC
350             IINC!  III
351             BSLB!  AA,ML,MM   ;AA=ML(!MM
352             IFNOT! AA,IF6
353             BEB!   MM,ML
354 IF6
355 WH2         GOSUB! GI
356             BNEB!  AA,CC,LF   ;AA=CC!=LF
357             IFNOT! AA,NDWH2
358             JR     WH2
359 NDWH2
360             JP     NDCH1
361 CS3         CASE!  OG,CS4     ;IF USING IGNORE, IGNORE
362             IFNOT! UG,IF7
363             JR     IF8
364 IF7         JP     LOC77
365 IF8
366             JP     NDCH1
367 CS4         REM               DEFAULT
368 LOC77
369             BIEB!  LS,III,CC  ;LS(III)=CC
370             IINC!  III
371             ISLI!  AA,ILM,III  ;AA=ILM('III
372             IFNOT! AA,IF9
373             MS!    'MACMEMXST'
374             GOSUB! CR
375             CLOSE! F1
376             STOP!  5
377 IF9
378             BINC!  ML
379 NDCH1
380             JP     WH1
381 NDWH1
382             BNEB!  AA,CC,EF   ;AA=CC'=EF
383             IFNOT! AA,IF10
384             MS!    'DEFN READ'
385             STOP!  2
386 IF10
387             CLOSE! F1
388             IECON! IBC,3
389 * ASSOCIATE FCB 3 WITH IBC
390             BECON! TR,'R'
391             OPEN!  F1,TR,IBC
392             IEI!   IED,III    ;IED=III ;END OF DEFS
393             MS!    'LOADED...'
394             IEI!   ITU,III
395             GOSUB! PN
396             MS!    '.BYTES FO'
397             MS!    'R DEFINES'
398             GOSUB! CR
399             IIEI!  ILP,INM,III  ;ILP(INM)=III
400             IEI!   ITU,INM
401             GOSUB! PN
402             MS!    '.MACROS..'
403             IEB!   ITU,MM      ;ITU=MM
404             GOSUB! PN
405             MS!    ' MIN LEN.'
406             GOSUB! CR
407             RET               ;ENDSUB RM
408 *
409 * Do carriage return
410 *
411 SUBCR        WRITE! NL
```

(Continued on top of page 32)

```
481             BEB!   AA,02
482             GOSUB! CD
483             IEB!   ITU,AA
484             JP     NDCH2
485 CS10        CASE!  OT,CS11    ;TURN ON TRACE MODE
486             BECON! UT,1       ;UT=+001
487             JR     NDCH2
488 CS11        REM               DEFAULT: FETCH UNIQUE #
489             IEI!   ITU,IUU    ;ITU=IUU
490             IINC!  IUU
491 NDCH2
492             IFNOT! UT,IF17
493             IEI!   III,ITU
494             GOSUB! PN
495             IEIB!  ITU,ISS,SP  ;ITU=ISS(SP)
496             GOSUB! PN
497             IEB!   ITU,SP     ;ITU=SP
498             GOSUB! PN
499             IEI!   ITU,III
500             GOSUB! CR
501 IF17
502             CHSON! 03         ;CHOOSE ON 03
503             CASE!  OC,CS12    ;CHAR OUTPUT
504             BEI!   AA,ITU
505             GOSUB! WA         ;WRITE AA INTO F2
506             JP     NDCH3
507 CS12        CASE!  OS,CS13    ;PUT ON STACK
508             BINC!  SP
509             BLEB!  AA,C4,SP
510             IFNOT! AA,IF18
511             MS!    'S OVERFLO'
512             GOSUB! CR
513             BEB!   SP,C4
514 IF18
515             IBEI!  ISS,SP,ITU
516             JP     NDCH3
517 CS13        CASE!  OP,CS14 PUT INTO PARAMETER LOCATION
518             IBEI!  IPR,DG,ITU
519             JP     NDCH3
520 CS14        CASE!  OA,CS15    ;ADD TO STACK
521             IFIB!  IAA,ISS,SP
522             IEIPI! IAA,IAA,ITU  ;IAA=IAA+ITU
523             IBEI!  ISS,SP,IAA
524             JP     NDCH3
525 CS15        CASE!  OR,CS16    ;SUBTRACT FROM STACK
526             IEIB!  IAA,ISS,SP
527             IEIMI! IAA,IAA,ITU  ;IAA=IAA-ITU
528             IBEI!  ISS,SP,IAA
529             JP     NDCH3
530 CS16        CASE!  OM,CS17    ;MULTIPLY BY BASE AND ADD
531             IEIB!  IAA,ISS,SP
532             ITMS!  IAA,IAA,I10  ;IAA=IAA*I10
533             IEIPI! IAA,IAA,ITU
534             IBEI!  ISS,SP,IAA
535             JP     NDCH3
536 CS17        CASE!  OH,CS18    ;OUTPUT HIGH BYTE
537             ITBB!  ITU,AA,BB  ;UNPACK(ITU,AA,BB)
538             GOSUB! WA         ;WRITE AA INTO F2
539             JR     NDCH3
540 CS18        GOSUB! WN         ;WRITE AS DECIMAL #
541 NDCH3
542             JP     IF14
543 IF13        REM               ELSE END OF ACTION SECTION
544             BEBI!  AA,LS,IMP
545             IFNOT! UN,IF19
546             BNEB!  BB,AA,RC   ;BB=AA!=RC
```

(Continued on next page)

Tinsource (Listing continued, text begins on page 25)

Listing Two *(Continued)*

```
547                 JR      IF20
548 IF19            BEB!    BB,C1
549 IF20            IFNOT!  BB,IF21
550                 GOSUB!  WA
551                 JR      IF22
552 IF21            BEB!    AA,NL
553                 GOSUB!  WA
554                 BEB!    AA,LF
555                 GOSUB!  WA
556 IF22
557 IF14
558                 IINC!   IMP
559                 JP      WH3
560 NDWH3
561                 BECON!  UT,0
562                 RET                 ;ENDSUB DM
563 *
564 * Write a byte to output
565 *
566 SUBWA          WRFIL!  AA,F2       ;WRITE AA INTO F2
567                 RET
568 *
569 * Get a char from input file
570 *
571 SUBGI          RDFIL!  CC,F1       ;READ CC FROM F1
572                 RET
573 *
574 * Get char & goto LOC88 on end
575 *
576 SUBGC          GOSUB!  GI          ;GET A CHAR
577                 BNEB!   AA,ER,C0
578                 IFNOT!  AA,IF22A
579                 GOTO!   88
580 IF22A          BEEB!   AA,CC,EF
581                 IFNOT!  AA,IF23
582                 GOTO!   88
583 IF23
584                 RET                 ;ENDSUB GC
585 *
586 * Converts number to series of ascii digit
587 *
588 SUBSD          ISLI!   AA,ITU,I00  ;AA=ITU<<I00
589                 IFNOT!  AA,IF24
590                 BECON!  BB,1
591                 IEMI!   ITU,ITU     ;ITU=-ITU
592                 JR      IF25
593 IF24           BECON!  BB,0
594 IF25
595                 IEEI!   AA,ITU,I00  ;AA=ITU==I00
596                 IFNOT!  AA,IF26
597                 BEB!    ND,C1
598                 BBEB!   DS,C0,ZR
599                 JP      IF27
600 IF26           BEB!    ND,C0
601 WH4            ISLI!   AA,I00,ITU
602                 IFNOT!  AA,NDWH4
603                 IDV!    IYY,ITU,I10 ;IYY=ITU/I10
604                 ITMS!   IAA,I10,IYY ;IAA=I10*IYY
605                 IEIMI!  IXX,ITU,IAA ;IXX=ITU-IAA
606                 IEI!    ITU,IYY
607                 BEI!    AA,IXX
608                 BEBPB!  AA,AA,ZR    ;AA=AA+ZR
609                 BBEB!   DS,ND,AA    ;DS(ND)=AA
610                 BINC!   ND
611                 JR      WH4
612 NDWH4
613 IF27
614                 BBEB!   DS,ND,CR
615                 BEBPB!  ND,ND,BB    ;MINUS SIGN FOR NEG INTEGER ONLY
616                 RET                 ;ENDSUB
617 *
618 * Write # into F2
619 *
620 SUBWN          GOSUB!  SD
621 WH5            IEB!    IAA,ND      ;WRITE OUT DIGITS
622                 ISLI!   AA,I00,IAA  ;AA=I00<<IAA
623                 IFNOT!  AA,NDWH5
624                 BDEC!   ND
625                 BEBB!   AA,DS,ND    ;AA=DS(ND)
626                 GOSUB!  WA
627                 JR      WH5
628 NDWH5
629                 RET                 ;ENDSUB
630 *
631 * Write # to terminal
632 *
633 SUBPN          GOSUB!  SD
634 WH6            IEB!    IAA,ND      ;IAA=ND
635                 ISLI!   AA,I00,IAA
636                 IFNOT!  AA,NDWH6
637                 BDEC!   ND
638                 BEBB!   AA,DS,ND
639                 WRITE!  AA
640                 JR      WH6
641 NDWH6
642                 WRITE!  BL
643                 RET                 ;ENDSUB
644 *
645 * Convert AA as a decimal digit
646 *
647 SUBCD          BLEB!   BB,ZR,AA    ;BB=ZR(=AA
648                 BLEB!   CC,AA,C9
649                 BEBAB!  BB,BB,CC    ;BB=BB&CC
650                 IFNOT!  BB,IF28
651                 BEBMB!  AA,AA,ZR    ;AA=AA-ZR
652                 RET                 ;RETURN
653 IF28           BEB!    AA,C0
654                 RET                 ;ENDSUB
655 *
656 * Convert AA as hex digit
657 *
658 SUBCH          BLEB!   BB,ZR,AA
659                 BLEB!   CC,AA,C9
660                 BEBAB!  BB,BB,CC
661                 IFNOT!  BB,IF29
662                 BEBMB!  AA,AA,ZR
663                 RET                 ;RETURN
664 IF29           BLEB!   BB,HA,AA
665                 BLEB!   CC,AA,HF
666                 BEBAB!  BB,BB,CC
667                 IFNOT!  BB,IF30
668                 BEBMB!  AA,AA,HA
669                 BEBPB!  AA,AA,CX
670                 RET                 ;RETURN
671 IF30           BEB!    AA,C0
672                 RET                 ;ENDSUB
673 *
(A:TINCMP)ff
```

(Continued on column 2)

End Listing

Simplified 68000 Mnemonics

by W.D. Maurer

If the 68000 is to become the most important new computer of the next five years, there will necessarily be a tremendous amount of interest in its assembly language. The profusion of features and facilities on the 68000, however, makes it very difficult to get a clear overall view of the instruction set. If one is to program reasonably well in assembly language, one must be able to lay out in one's mind which facilities are there and which are not. With the 68000, it often seems as if one is let loose upon a tropical island, full of flowers of dazzling beauty but so much underbrush that it seems impossible to scale its central mountain peak and survey the totality of the landscape.

The main problem is with the mnemonics. This is not to say that the mnemonics show any glaring examples of bad design. They are serviceable, as far as they go; it is simply that it is possible to do much better. The purpose of this paper is to *propose* a new collection of mnemonics for the 68000. This is not to say that an assembler has been written, using these mnemonics (at the moment the author does not have access to a 68000-based system). Rather, the purpose here is to show the many advantages of this set of mnemonics over Motorola's. There is precedent for such redesigns; for example, the mnemonics of the UNIVAC 1100 series were redone by CSC in 1962 in connection with CSC's design of the EXEC II operating system for the 1100 series (a project in which this author took part).

The new mnemonics are given in Table I. The notation used in Table I is explained in Table II. It is important to note that the new mnemonics allow the entire instruction repertoire of the 68000 – together with very brief explanations of all instructions – to be presented upon one single-spaced page, with all notational conventions laid out on another single-spaced page.

A program selected at random from a recent book on the 68000[1] is presented in Table III with both the old and new mnemonics. Except for labels, which remain the same in the two cases, the total number of keystrokes of non-blank characters has been reduced from 196 to 98 – a reduction of 50%. The simplicity of understanding for the new mnemonics makes it easy, in fact, to spot a bug in the program of Table III. This will be explained in more detail at the end of the paper.

The basic ideas behind the new mnemonics are as follows:

(1) *The data registers and the address registers all receive new designations.* The data registers are now called the A, B, C, D, E, F, G, and H registers, each of which is 32 bits long. The address registers are now called the S, T, U, V, W, X, Y, and Z registers, and each of these is 32 bits long. In particular, the S register is the stack pointer, known conventionally as A7. (The original designations for the registers A through H are D0 through D7, and those for S through Z are A7 through A0, in reverse order, with A0 corresponding to Z.)

(2) *There are no suffixes B, W, and L for byte, word, and long word operations.* As we shall see, most instructions do not need such suffixes because they operate on data which has been defined as byte,

G	OPERATION	DESCRIPTION
3	A v,i	Add i to v
1	Aa v	Add v to a (v ≠ lt)
	ABab	Add BCD lb to la
	ABtu	Add BCD lMDu to lMDt
4	AMa v	Add a to v (memory)
4	ASd v2	Arith. shift v2 by 1
	ASd ma,sc	Arith. shift ma by sc
1	At vx	Add vx to t
	AXmab	Add extended mb to ma
	AXmtu	" " mMDu to mMDt
2	An v	Add n to v (v ≠ lt)
	B p	Branch to p
	Baf i2	Test bit i2 of a, op f
	Bafb	" " y (in b) " " "
	BaT i2	Test bit i2 of a
	BaTb	Test bit y (in b) of a
	Bcc p	Branch to p on cc
4	Bf v1,i2	Test bit i2 of v1, op f
4	Bfa v1	" " y (in a) " " " "
	BS p	Branch to subroutine p
5	BT v1,i2	Test bit i2 of v1
5	BTa v1	Test bit y (in a)of v1
3	C v,i	Compare i with v
1	Ca v	Compare a with v(v≠lt)
6	CKa v2	Check 2a bounded by v2
	Cmtu	Compare mMtI with mMuI
1	Ct vx	Compare t with vx
	Dcca p	Bcc β;a←a-1;a=-1?→p;β:
6	DSa v2	Divide signed 4a/v2
6	DUa v2	Divide unsigned 4a/v2
	Ea	Exchange halves of a
	Egh	Exchange 4g with 4h
	FAt -i2	Frame allocation (iMt)
	FDt	Frame deallocation
	H i2	Halt (status reg.←i2)
7	J v	Jump to v
7	JS v	Jump to subroutine v
7	LAt v	Load t with addr. of v
4	LSd v2	Logical shift v2 by 1
	LSd ma,sc	Logical shift ma by sc
1	M v,w	Move w to v (w≠lt)
1	Ma v	Move v to a (v≠lt)
6	MCC v2	Move v2 (byte 2) to CC
	MKa k	Move signed const. k→a
1	MMa v	Move a to v (memory)
1	MMt vx	Move t to vx (memory)
9	MR rs,vx	Move vx → registers rs
	MRxt rs	Move regs(=MR rs,xMtI)
6	MSa v2	Multiply signed 2a*v2
3	MSM v2	Move status reg. to v2
P6	MSR v2	Move v2 to status reg.
P	MSt	Move t to S
1	Mt vx	Move vx to t
P	MtS	Move S to t
6	MUa v2	Multiply uns. 2a*v2
3	N v,i	Logical AND i to v
6	Na v	Logical AND v to a
	NBa	Negate BCD la
4	NBM v1	Negate BCD v1 in mem.
	NCC i1	Logical AND i1 to CC
	NGma	Negate ma (2's comp.)
4	NGM v	Negate v in memory
3	NMa v	AND a to v (memory)
	NO	No operation
P	NSR i2	Logical AND i2 to SR
	NTma	NOT ma (ones' compl.)
4	NTM v	NOT v in memory
	NXma	Negate extended ma
4	NXM v	" " v in memory
3	O v,i	Logical OR i to v
6	Oa v	Logical OR v to a
	OCC i1	Logical OR i1 to CC
3	OMa v	OR a to v (memory)
P	OSR i2	Logical OR i2 to SS
7	PA v	Push address of v
1	PH v	Push v (= M mMDS,v)
1	PL v	Pull v (= M v,mMSI)
	PMxa Q(t)	xa←Q(t) alternate bytes
	Pxa Q(t)	Q(t)←xa " " (periph.)
	R	Return from subroutine
	RE	" " excptn. (pull SR)
8	RM rs,vx	Regs to memory rs → vx
	RMxt rs	Regs-mem.(=RM rs,xMDt)
	ROd v2	Rotate v2 by 1
	ROd ma,sc	Rotate ma by sc
	RR	Ret.&restore (pull CC)
	RS	Reset external devices
	RXd v2	Rot. extended v2 by 1
	RXd ma,sc	Rot. extended ma by sc
3	S v,i	Subtract i from v
1	Sa v	Subtr. v from a (v≠lt)
	SBab	Subtr. BCD lb from la
	SBtu	" " lMDu from lMDt
3	Scc v1	Set v1=-1 (if cc) or 0
4	SMa v	Subtr. a from v (mem.)
1	St vx	Subtract vx from t
	SXmab	Subtr. extended ma-mb
	SXmtu	Subtr. ext. mMDt-mMDu
2	Sn v	Subtr. n from v (v≠lt)
3	TE v	Test v, set flags N&Z
	TR vn	Trap, using vector vn
3	TS v1	TE v1; v1(hi-order) =1
	TV	BVC β; TR 7; β:
3	X v,i	Exclusive OR i to v
	XCC i1	Exclusive OR i1 to CC
3	XMa v	Excl. OR a to v (mem.)
P	XSR i2	Exclusive OR i2 to SR
	Xxa	Extend sign a (to xa)
3	Z v	Zero (clear) v

Table I

W. D. Maurer, George Washington University, S.E.A.S., Washington, DC 20052.

word, or long word data, which determines the type of the operation automatically. In cases where the length must be given, however, it is given in bytes as 1, 2, or 4. This has the immediate advantage of avoiding the non-standard term "word." (A word has 32 bits, not 16, on the IBM 360 and 370, for example.)

The length designation appears in the following places:

(a) *In certain of the mnemonics.* The instruction whose Motorola mnemonic is ADDX.L D4,D3 (Add Extended, Long Word, register D4 to register D3) is simply AX4DE in the new mnemonics. Here D and E denote the D register (formerly D3) and the E register (formerly D4) and the 4 denotes, very simply, "four bytes." Note that the order is "destination first, then source"; see point (3) below to appreciate how much simpler this makes a great number of the mnemonics. The corresponding byte and word comparison instructions ADDX.B D4,D3 and ADDX.W D4, D3 become AX1DE and AX2DE respectively.

```
a = A, B, C, D, E, F, G, or H (data registers)
b = A, B, C, D, E, F, G, or H (data registers)
CC = condition code register (rightmost byte of SR)
cc = one of the following forms:
     T    true              VC   overflow clear
     F    false             VS   overflow set
     HI   high              PL   plus
     LS   low or same       MI   minus
     CC   carry clear       GE   greater or equal
     CS   carry set         LT   less than
     NE   not equal         GT   greater than
     EQ   equal             LE   less or equal
d = L (left shift) or R (right shift)
e = label, label±j, or other address expression (length defined
    in definition of given label) or mBlabel (length m)
f = N (op=AND with 0), O (op=OR with 1), or X (op=XOR with 1)
g = A, B, C, D, E, F, G, H, S (stack pointer), T, U, V, W,
    X, Y, or Z (data register or address register); length = 4
h -- same as g (second register in exchange instruction)
i = y (constant) or e (address of e) (immediate data)
i1 -- same as i but length must be 1 (immediate data)
i2 -- same as i but length must be 2 (immediate data)
j = $hh... or dd... or %bb... where the h's are hexadecimal digits,
    the d's decimal digits, and the b's binary digits (constant)
k -- +k is +0 through +127; -k is -1 through -128 (constant in move
    instruction MK; 8-bit displacement in based indexed modes)
m -- 1, 2, or 4 (length, or number of bytes in register or memory)
n -- 1, 2, 3, 4, 5, 6, 7, or 8 (additive or subtractive constant)
p = q3, but *-32766 ≤ p ≤ *+32769 (relative address; *=this loc.)
q = e (data) or mBj (data, length m, at address j) (memory loc.)
q2 -- same as q but address must have length 2 (memory location)
q3 -- same as q but address must have length 3 (memory location)
r = q3, but *-126 ≤ r ≤ *+129 (relative address; *=this location)
rs = z1/z2/... where each zk = g or g-h (g, h as above; g-h means
     registers g, g+1, ..., h) (multiple registers for MR and RM)
sc = 1, 2, 3, 4, 5, 6, 7, 8, A, B, C, D, E, F, G, H (shift count
     -- in rightmost 6 bits of register A-H if this is specified)
SR = status register (length 2; CC = rightmost byte of SR)
t = S (stack pointer), T, U, V, W, X, Y, or Z (address registers)
u = S (stack pointer), T, U, V, W, X, Y, or Z (address registers)
v = one of the following forms (variable in register or memory):
    #i (immediate)                      (Groups 1 and 6 only)
    q3 (absolute long)                  (All groups)
    q2 (absolute short)                 (All groups)
    ma (data register direct)           (Groups 1, 2, 3, and 6 only)
    mt (address register direct)        (Groups 1 and 2 only)
    mMt (address register indirect)     (All groups)
    mMtI (post-increment)               (All groups except 7 and 8)
    mMDt (pre-decrement)                (All groups except 7 and 9)
    mMtg or mMtg±k (based indexed short) (all groups)
    mMt4g or mMt4g±k (based indexed long) (all groups)
    q2(t) or mMt±q2 (based)             (All groups)
    p (relative)                        (Groups 1 and 5 through 9 only)
    r(t) or r(4t) (relative indexed)    (Groups 1 and 5 through 9 only)
vm -- like v but with length m (variable in register or memory)
vn -- 0 through 15 (vector number in trap instruction)
w -- like v in M v,w but group 3 and length must equal length of v
x -- like m, but x ≠ 1 (vx is like v but with length 2 or 4)
y = mBj (length m) or j (length as small as possible) (integer)
```

Table II

(b) *Preceding register designations.* The instruction MOVE.L A3,A6 (old mnemonics) moves A3 to A6. Renaming these registers W and T respectively, we can write M 4T,4W (reversing the order as before). It is simpler, however, to write MT 4W where MT means "move to the T register" and 4W means "four bytes of the W register." The corresponding byte and word move instructions MOVE.B A3,A6 and MOVE.W A3, A6 become MT 1W and MT 2W respectively.

(c) *As part of more general addressing modes.* Address register indirect addressing, involving the X register, in a one-byte instruction, would be denoted by 1MX (*one* byte of *memory* indexed by *X*.) The same thing, with auto-increment or auto-decrement, would be 1MXI or 1MDX respectively. (Note that the I, for "increment," *follows* the register name, whereas D, for "decrement," *precedes* it; this resembles the placement of + and - in the Motorola mnemonics, and is done for the same reason – because incrementation follows the reference to memory, while decrementation precedes it.) Even the most complex mode, namely based indexed long, is straightforward; thus 2MU4E-9 means "*two* bytes of *memory*, indexed by the *U* register and by all *four* bytes of the *E* register, *minus* the displacement *9*." The same thing in Motorola mnemonics would be -9(A5,D4.L) – over 50% more characters (not counting the extra .W on the instruction mnemonic) and lacking the straightforward readability of the new expression.

(3) *Most operation codes have only one argument.* In fact the *only* operation codes with two arguments are moves, shifts, and immediate addressing instructions. Thus M KPREV,KCURR sets KPREV (in memory) equal to KCURR (in memory); A FCOUNT,5 adds 5 to FCOUNT in memory; ASL 1H,4 shifts 1H (*one* byte of the *H* register, formerly known as D7) left arithmetically by 4. (If the mnemonic variations are counted separately – if MA through MH are counted as eight instructions instead of one, for example – then almost 75% of the mnemonics have *no* arguments; about 25% have one; and only 2% have two.)

The destination register in a great number of the arguments is incorporated into the mnemonic. For the C register, for example (formerly D2), we have MC (move to C, AC, SC, CC (add, subtract, and compare), MSC, MUC, DSC, DUC (multiply and divide by C, signed and unsigned), NC (AND), OC (OR), and so on. For a few "operate to memory instructions" such as AMC (add to memory), SMC, NMC, and OMC, the C register is the source, rather than the destination. There are also EC (exchange halves of C), ECD (exchange C and D), NG4C (set C to its two's complement), X2C and X4C (extend sign to two bytes or four bytes of C), and other such exceptional cases.

(4) *All operations on data have lengths determined by the data unless otherwise directed.* If AMOUNT is a 32-bit variable, then MF AMOUNT moves AMOUNT to the F register, and is a long-word (32-bit) operation. To move only one or two bytes of AMOUNT to the F register, one uses MF 1BAMOUNT or MF 2BAMOUNT respectively.

Tables I and II are meant to be used in tandem. As an example, consider the second entry in Table I, namely A*a v* (Add *v* to *a*). By consulting Table II, one learns that *a* may be A, B, C, D, E, F, G, or H, and that *v* may have one of several forms. Let us consider the form *mt*. Here *m* is 1, 2, or 4, but in the description of A*a v* we find the designation "*v*≠1*t*"; that is, *v* cannot be of the form 1*t*, so that *m*, here, must be 2 or 4, while *t* is S, T, U, V, W, X, Y, or Z (again by reference to Table II). Thus there are the following possibilities (for example):

AA 2X
(Add to A from two bytes of X)
AD 4T
(Add to D from four bytes of T)

for a total of 128 (8 x 2 x 8) possibilities. Another form of *v*, in this same instruction, is *q*3, described as "same as *q* but address must have length 3." So this is a 24-bit (or three-byte) address, and we have the following further possibilities for *v*:

q = *e* = label
(label of location with three-byte address)
q = *m*B*j* = 4B$hh. . .
(length 4, but *address* still has length 3)
q = *e* = label±*j* = label+dd. . .
q = *e* = *m*Blabel
(data has length *m*; address has length 3)

This means that if NST is a two-byte variable, ECOUNT a four-byte variable, and F7 an array of single-byte variables, we could also write

AF ECOUNT
(Add to four bytes of F from ECOUNT)
AD 4B$FC
(Add to D from four bytes starting at address $FC)
AG F7+50
(Add to one byte of G from byte 50 of F7)
AA 1BNST
(Add to one byte of A from one byte of NST)

Still another form of *v*, in this same instruction, is #*i*, for immediate data. By Table II, we can have *i* = *y* = *m*B*j* = 4B$hh. . . which gives us the possible instruction

AD #4B$FC
(Add $FC to four bytes of D)

Note the two meanings of the 4 in 4B$FC – in this instruction it means "the constant $FC as a four-byte constant," whereas in AD 4B$FC it means "four bytes of memory."

The column headed G in Table I is the *group*. Instructions belong to groups, and each group can use only certain of the *v* options as specified in Table II. For example, the instruction LA*t v* belongs to group 7. This means that *v* can be of the form *r*(*t*) or *r*(4*t*), since these are specified in Table II as "groups 1 and 5 through 9 only." Thus if WTABLE is a suitable relative address, then each of the following is *legal*:

LAT WTABLE(Z)
(Load address of WTABLE plus Z into T)
LAV WTABLE(4W)
(Load address of WTABLE plus W into V)

where two bytes of Z, and four bytes of W, are used. On the other hand, *v* cannot be of the form *m*M*t*I or *m*MD*t*, since these are specified in Table II as "all groups except 7 and 8" or "all groups except 7 and 9." Thus each of the following is *illegal*:

LAT 2MZI
LAV 4MDW

It should be noted that the length (1, 2, or 4) is meaningless in the case of the specific instructions LA*t*, PA*t*, J, and JS. Thus in the program of Table III, LAY

```
        OLD MNEMONICS                          NEW MNEMONICS

PGM_9_4A MOVEA.L  LIST,A0          PGM_9_4A  MZ    LIST
         CLR.W    D0                          Z     2A
         MOVE.B   (A0)+,D0                    MA    1MZI
         LEA      -1(A0,D0.W),A1              LAY   1MZA-1
SORT     CLR.W    D1               SORT       Z     2B
         MOVEA.L  A0,A2                       MX    4Z
NEXT     MOVE.B   (A2)+,D0         NEXT       MA    1MXI
         CMP.B    (A2),D0                     CA    1MX
         BCC.S    NSWITCH                     BCC   NSWITCH
         MOVE.B   (A2),D1                     MB    1MX
         MOVE.B   D1,-1(A2)                   MMB   1MX-1
         MOVE.B   D0,(A2)                     MMA   1MX
         ADDQ.W   #1,D1
NSWITCH  CMPA.L   A2,A1            NSWITCH    CY    4X
         BHI      NEXT                        BHI   NEXT
         TST.W    D1                          TE    2B
         BNE      SORT                        BNE   SORT
         RTS                                  R

                          REGISTERS

OLD   D0 D1 D2 D3 D4 D5 D6 D7 A0 A1 A2 A3 A4 A5 A6 A7

NEW    A  B  C  D  E  F  G  H  Z  Y  X  W  V  U  T  S
```

Table III

1MZA-1 means "*L*oad *A*ddress, into *Y*, of the *memory* word indexed by *Z* and by *A*, *minus 1*." The 1 before the M means nothing, but it must be present (although 2 or 4 would do just as well), since LAY MZA-1 would load the address of a variable called MZA, minus 1. (Privileged instructions are group P.)

Examples of the *v* options which start with a length (1, 2, or 4) are as follows:

(1) 1C (one byte of the C register; the other three bytes are ignored)

(2) 2V (two bytes of the V register; extended through the other two bytes)

(3) 1MU (one byte in memory, indexed by U – that is, the address of this byte is contained in U)

(4) 2MWI (two bytes in memory, indexed by W, and then increment W by 2 after making the reference)

(5) 4MDT (four bytes in memory, indexed by T, and decrement T by 4 *before* making the reference)

(6) 2MUE (two bytes in memory, indexed by U and by E – that is, the address of this byte is the sum of U and two bytes of E)

(7) 1MWX (one byte in memory, indexed by W and by X)

(8) 4MZ4G (four bytes in memory, indexed by Z and all four bytes of G)

(9) 4MXY+8 (four bytes in memory, indexed by X and by Y, plus 8 – that is, the address of this byte is the sum of X and two bytes of Y, plus 8)

(10) 1MZ4S-16 (one byte in memory, indexed by Z and by all four bytes of the stack pointer S, minus 16)

Table IV gives examples of all the instructions in both the old and the new mnemonics; Table V gives a more expanded explanation of the program of Table III. We may note from Table V that the use of the new mnemonics makes it much easier to spot a bug in this sorting program, intended to sort a table of unsigned, single-byte quantities into *descending* order. The B register (formerly D1) is being used as the exchange flag: set to zero (Z 2B), incremented to show the exchange (A1 2B), and tested at the end TE 2B). But it is also used as a temporary register in moving 1MX to 1MX-1 (in the instructions MB 1MX and MMB 1MX-1). If the bug is not fixed, the instruction A1 2B might increment -1 to zero, making it look like the exchange flag is still clear.

An example which shows the bug is that of a table of size greater than $82 (that is, hexadecimal 82), which is already sorted in descending order except that its last two bytes are $FF (and no other byte is $FF). In the given program, these final two bytes move up by one position in each pass through the table; and only two exchanges (the ones involving these two bytes) are made in each pass, since the table is otherwise already sorted in the proper order. Therefore, MB and A1 are executed only twice in each pass through the table. In the first pass, $FF is moved into B and this is then incremented by 1 *as a two-byte quantity*, producing $100. Subsequently, $FF is moved into the *rightmost byte* of B, leaving the rest of B undisturbed; this produces $1FF, which is then incremented to $200. Thus the final value of B is $200 at the end of the first pass, and, in general, it is 2 x $*k*00 at the end of the *k*th pass. At the end of the 128th pass, therefore, B will be zero; the TE then tests for

MNEMONIC	EXAMPLE	MOTOROLA MNEMONIC
A v,i	A 2MZI,1	ADD.W #1,(A0)+
Aa v	AA 4MDY	ADD.L -(A1),D0
ABab	ABGH	ABCD D7,D6
ABtu	ABWX	ABCD -(A2),-(A3)
AMa v	AMB BYT1	ADD.B D1,BYT1
ASd v2	ASR WD1	ASR WD1
ASd ma,sc	ASL 2C,4	ASL.W 4,D2
At vx	AY 4MV	ADD.L (A4),A1
AXmab	AX1DE	ADDX.B D4,D3
AXmtu	AX2UZ	ADDX.W -(A0),-(A5)
An v	A4 4F	ADDQ.L #4,D5
B p	B LABEL1	BRA LABEL1
Baf i2	BGN 8	BCLR #8,D6
Bafb	BHOF	BSET D5,D7
BaT i2	BET $10	BTST $10,D4
BaTb	BDTC	BTST D2,D3
Bcc p	BEQ BETA	BEQ BETA
Bf v1,i2	BX M9,6	BCHG 6,M9
Bfa v1	BNB 1MTI	BCLR D1,(A6)+
BS p	BS PROG3	BSR PROG3
BT v1,i2	BT M8,3	BTST 3,M8
BTa v1	BTA 1MDZ	BTST D0,-(A0)
C v,i	C 4MSI,5	CMP.L #5,(A7)+
Ca v	CD 1A	CMP.B D0,D3
CKa v2	CKB WD1	CHK WD1,D1
Cmtu	C4YZ	CMPM.L (A0)+,(A1)+
Ct vx	CX 2W	CMP.W A3,A2
Dcca p	DCSC L19	DBCS D2,L19
DSa v2	DSD WD1	DIVS WD1,D3
DUa v2	DUE 2MU	DIVU (A5),D4
Ea	EF	SWAP D5
Egh	ESA	EXG A7,D0
FAt -i2	FAZ -23	LINK A0,-23
FDt	FDT	UNLK A6
H i2	H $F0F0	STOP $F0F0
J v	J 1MYD-8	JMP -8(A1,D3)
JS v	JS 1MX4G	JSR (A2,D6.W)
LAt v	LAW 1MVI	LEA (A4)+,A3
LSd v2	LSL WD1	LSL WD1
LSd ma,sc	LSR 1F,H	LSR.B D7,D5
M v,w	M LW1,J3	MOVE.L J3,LW1
Ma v	MC 2,#$C	MOVE.B #$C,D2
MCC v2	MCC $1A	MOVE $1A,CCR
MKa k	MKH -100	MOVEQ -100,D7
MMa v	MMA WD2	MOVE.W D0,WD2
MMt vx	MMV 4MU	MOVE.L A4,(A5)
MR rs,vx	MR T,LW1	MOVEM.L LW1,A6
MRxt rs	MR4Z A/H	" (A0)+,D0/D7
MSa v2	MSB 2MDW	MULS -(A3),D1
MSM v2	MSM 2MVI	MOVE SR,(A4)+
MSR v2	MSR 2MU	MOVE (A5),SR
MSt	MSX	MOVE A2,USP
Mt vx	MY 2MZ	MOVE.W (A0),A1
MtS	MUS	MOVE USP,A5
MUa v2	MUC WD3	MULU WD3,D2

MNEMONIC	EXAMPLE	MOTOROLA MNEMONIC
N v,i	N 4MDY,8	AND.L #8,-(A1)
Na v	NB 1MXI	AND.B (A2)+,D1
NBa	NBC	NBCD D2
NBM v1	NBM 1MZT	NBCD 0(A0,A6)
NCC i1	NCC $1A	AND.B #$1A,SR
NGma	NG1A	NEG.B D0
NGM v	NGM 2MUD	NEG.W 0(A5,D3)
NMa v	NMD 4MDS	AND.L D3,-(A7)
NO	NO	NOP
NSR i2	NSR $F0F	AND.W #$F0F,SR
NTma	NT2E	NOT.W D4
NTM v	NTM BYT1	NOT.B BYT1
NXma	NX4F	NEGX.L D5
NXM v	NXM 1MS	NEGX.B (A7)
O v,i	O 2G,$AA	OR.W #$AA,D6
Oa v	OH BYT1	OR.B BYT1,D7
OCC i1	OCC 8	OR.B #8,SR
OMa v	OMB 4MUI	OR.L D1,(A5)+
OSR i2	OSR $200	OR.W #$200,SR
PA v	PA 1MV	PEA (A4)
PH v	PH WD1	MOVE.W WD1,-(A7)
PL v	PL 4A	MOVE.L (A7)+,D0
PMxa Q(t)	PM2B J(Z)	MOVEP.W D1,J(A0)
Pxa Q(t)	P4C J(Y)	MOVEP.L J(A1),D2
R	R	RTS
RE	RE	RTE
RM rs,vx	RM T,LW1	MOVEM.L A6,LW1
RMxt rs	RM4Z T-Y	" A1-A6,-(A0)
ROd v2	ROR WD1	ROR WD1
ROd ma,sc	ROL 1D,C	ROL.B D2,D3
RR	RR	RTR
RS	RS	RESET
RXd v2	RXL WD1	ROXL WD1
RXd ma,sc	RXR 4E,6	ROXR.L 6,D4
S v,i	S 1MW,32	SUB.B #32,(A3)
Sa v	SA 2MX4G-8	SUB.W -8(A2,D6.L),D0
SBab	SBFE	SBCD D4,D5
SBtu	SBZY	SBCD -(A1),-(A0)
Scc v1	SGT BYT1	SGT BYT1
SMa v	SME 4MWH	SUB.L D4,0(A3,D7)
St vx	SU 2F	SUB.W D5,A5
SXmab	SX4HB	SUBX.L D1,D7
SXmtu	SX1US	SUBX.B -(A7),-(A5)
Sn v	S7 2MTI	SUBQ.W #7,(A6)+
TE v	TE 1MV4X	TST.B 0(A4,A2.W)
TR vn	TR 4	TRAP 4
TS v1	TS 1G	TAS D6
TV	TV	TRAPV
X v,i	X 1MW,$FF	EOR.B #$FF,(A3)
XCC i1	XCC $2	EOR.B #$2,SR
XMa v	XMD LW1	EOR.L D3,LW1
XSR i2	XSR $400	EOR.W #$400,SR
Xxa	X4C	EXT.L D2
Z v	Z 1MZ4Y+2	CLR.B 2(A0,A1.W)

Table IV

```
PGM_9_4A  MZ    LIST      (Move LIST, which is a 4-byte quan-
                           tity, to the Z-register)
          Z     2A        (Zero, or clear, two bytes of the
                           A-register)
          MA    1MZI      (Move, to the A-register, one byte of
                           memory, indexed by the Z-register,
                           and then increment the Z-register by 1)
          LAY   1MZA-1    (Load into the Y-register the address
                           in memory indexed by the Z register
                           and by the A register, minus 1)
SORT      Z     2B        (Zero, or clear, two bytes of the
                           B-register)
          MX    4Z        (Move, to the X-register, four bytes
                           of the Z-register)
NEXT      MA    1MXI      (Move, to the A-register, one byte in
                           memory, indexed by the X-register, and
                           then increment the X-register by 1)
          CA    1MX       (Compare the A-register with one byte
                           in memory, indexed by the X-register)
          BCC   NSWITCH   (Branch on carry clear to NSWITCH)
          MB    1MX       (Move, to the B-register, one byte in
                           memory indexed by the X-register)
          MMB   1MX-1     (Move to memory, from the B-register,
                           to one byte in memory, indexed by the
                           X-register minus 1)
          MMA   1MX       (Move to memory, from the A-register,
                           to one byte in memory, indexed by the
                           X-register)
          A1    2B        (Add 1 to two bytes of the B-register)
NSWITCH   CY    4X        (Compare the Y-register with four bytes
                           of the X-register)
          BHI   NEXT      (Branch on high, i. e., Y > X, to NEXT)
          TE    2B        (Test two bytes of the B register)
          BNE   SORT      (Branch to SORT on unequal to zero)
          R               (Return from subroutine)


                    REGISTERS


OLD   D0 D1 D2 D3 D4 D5 D6 D7 A0 A1 A2 A3 A4 A5 A6 A7

NEW    A  B  C  D  E  F  G  H  Z  Y  X  W  V  U  T  S
```

Table V

zero; the BNE does *not* branch back; and the sort terminates with the two $FF bytes in positions 129 and 130 from the end, instead of in the first two positions, where they belong.

References

[1] Kane, G., D. Hawkins, and L. Leventhal, *68000 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, 1981.

DDJ

by Gene Head

Mail received after the first CP/M Exchange was encouraging. There is heavy interest in modem communication and program exchange. The most often asked question was, "How do I get started using my modem?" Well, it may be as simple as one, two three. I say *may* because each modem installation is different. If you have a basic understanding of the following three steps, you should be able to get signed-on to a Remote CP/M (RCP/M) System using the MBOOT3 program (listing begins on page 47).

1. Modem Status
2. Modem Input
3. Modem Output

If you understand how the CP/M functions CONSOLE STATUS, CONSOLE INPUT and CONSOLE OUTPUT operate, you also understand how the modem should operate. (Note: CONSOLE routines use only seven bits of data and the MODEM routines *must* use all eight bits.) Don't be put off if you fail to understand how to get the CP/M CONSOLE functions to operate. You'll need special assistance to get your modem functional, but I have a plan for you too!

MBOOT3 is a simple, stripped-down, receive-only version of Ward Christensen's MODEM program. The idea is to get MBOOT3 operating and use it to download (receive) a full MODEM program from an RCP/M system. If you have a modem, the software to simulate a terminal *and* the ability to "capture" received data into a file, just log on to an RCP/M and command the remote computer to list the MBOOT3 file using the CP/M command TYPE. Otherwise, spend an hour or so and type in MBOOT3.

Customize the hardware-dependent equates, assemble, and load MBOOT3. Follow the instructions on the facing page, remembering that each RCP/M system may be slightly different. Use your head. Don't be discouraged. Try to get the SYStem OPerator's help by using the CHAT command, or try ringing his bell by typing control-G. You can't hurt the remote system, and every RCP/M I've ever used has been very forgiving of errors and even untimely disconnections.

Modem Hint # 1

You will probably need cooperation from someone with a modem to test your system out before making long-distance calls to RCP/M's. If you can't get that special help, consider this: colleges with computer departments usually have a modem telephone for remote users. Without an account number and/or password you can't do much but you can at least verify that the basic communication, modem-to-modem, is operating. Execute the MBOOT3 program as directed in the source listing and then access any modem line by telephone. Try a few carriage returns to get the remote system "talking" to your system. Words like FULL and HALF and HELP might get results. Check it out. . .

Special Assistance

Here is where the novice can be helped by the experienced modem user. If you have the MODEM program working on your system, send me a card with all the details about your specific hardware, the source listing of *just the hardware-dependent equates* and any unique hardware-dependent code, and how they are configured for your set-up.

Likewise, if you have the hardware but don't know exactly how to customize the software, send me a card with your specific hardware specifications. (Keep it simple!) I'll match your request with a known working system and send you the customization details. If I can't get an exact match, I'll send along the best data I have available. *Be sure to enclose an SASE with your request.* I'll do free photocopying (but no free postage) as long as this doesn't get too involved. . .

If you can offer help or need help, let the "CP/M Exchange" know! Be sure

Downloading MODEM Using MBOOT3

First, execute MBOOT3 as instructed in the source listing (begins on page 47). It will sign on, but there will be no screen activity after the sign-on messages. In the TERMINAL mode, all displayed console characters come from the remote system. Even keys from your keyboard are "echoed" back, so if there is no remote system to echo back the keystrokes, you won't see them on your CRT.

Now, call a remote CP/M System (see list on page 44), when you hear the remote modem carrier tone, connect your modem to the phone line. The two modems are now connected and your computer is "talking" to the remote computer. Press the Return key once a second until you see the remote computer's sign-on message, usually asking how many nulls you need. (Answer 0 to the nulls question if you're using a video terminal.)

Follow the host computer's instructions until you finally reach the CP/M operating level with the expected A>. You are now in control of the remote machine.

To find the MODEM program type:

DIR MODEM *.*

This will show you the exact name of the MODEM files available. You should see in the listing the MODEM.ASM file or some similar source code file.

To receive the file type:

XMODEM filename.type

The remote system should tell you that XMODEM is ready to send the requested file, then it will wait. There will be no further screen activity from the host computer. It will look as though you have lost control, but don't fret. The host is waiting for your MBOOT3 program to send special protocol characters to begin the transfer and won't respond to your keyboard characters until the transfer is completed or aborted.

Now that you know the host is ready to send and is waiting for you to be ready to receive, press your ESCape key. This key tells MBOOT3 to receive the file the host is sending.

Most CP/M commands are available to remote users, so use the remote system just as you would your local computer. Be sure to sign off by typing BYE and disconnect your modem from your phone line, if it isn't automatically disconnected by your software.

As with any new operation, practice and experience are the best teachers.

to include your disk format and complete modem configuration. Send your complete name, address, city, state, zip, and phone with area code.

Exchange Mail

The first response to the CP/M Exchange came from *DDJ* contributor Bob Blum. Bob suggested electronic mail for communication. If the interest is there, we might be able to compile a directory of those using this form of communication. I will make my RCP/M system available if there is serious interest. Anyone else have ideas along these lines? Bob is also preparing a contribution on keyboard buffering in a modified CCP for CP/M that will be included in a future "CP/M Exchange."

The most specific letter seeking help came from Charles Henderson of Midland, Texas. Charles is interested in why SPEED and FAST won't work with CP/M 2.x. He suspects the track buffering is involved but doesn't understand all the implications. A definitive article on track and sector skewing, interleaving, sector tables, etc. would be most appropriate. If you have a solid understanding of these principles, write a paper and send it in! Share with others what you have learned.

Dana Trout of Goleta, California, notes that CP/M Function #37 does not work as described in the manual. This function to reset the disk drives will reset all drives *except* the current default drive. If the current default drive is set to R/O and function thirty-seven is executed, the default drive is still in R/O! Anyone have a fix and/or explanation for this?

Roland Lupient, KB9RR, has ham radio gear, an Altair 8800 and CP/M User's Group volume 41. He would like to hear from anyone who can help interface the ham gear and computer (address: 1953 Graham Lane, Mosinee, WI 54455).

Finally, William Burnett of Sinton, Texas, wonders if there are any Superbrain users out there who have access to the CP/M User's Library? The same question can be asked about North Star, Osborne, and all the other five-inch formats. If you can copy the CP/M User's Library to any of the many five-inch disk formats, let me know. I suspect you will be in great demand, and while you may not make a lot of money at this, a couple of bucks copying fee seems reasonable.

That covers it for now. Every letter to the "CP/M Exchange" has received either a personal response or was noted in this column. If you have something to contribute, a question or comment, let's hear from you!

▶▶J

Remote CP/M Software Exchange Systems
List # 24
Last Revised March 21, 1982

List #24 revised and updated courtesy of Hyde Park RCPM, CP/M-Net, and Mississauga RCPM.

This is part of a CP/M file called RCPMLIST.xx. It is found on most RCP/M systems. The systems presented here are representative of what you can expect to find. Each was "up" as of February, 1982, unless otherwise noted. Log on to any system and then get a copy of this complete file for an accurate directory.

This is a partial list.

California

(Best)
Bakersfield, CA
CP/M-Net™, (805) 527-9321, Kelly Smith
No A.L.D.S.; 1900-2300 (PST) Monday-Friday, 1900 Friday to 0700 Monday
110-600+ baud
20 Mb of files on 2 hard disks (=8 logical disks)
System now includes SIG/M Vol. 1-10 = E:, SIG/M Vol. 11-20 = F:, SIG/M Vol 21-25 = G:. XMODEM 'DISKMENU.DOC' for entire system directory (over 2100 files now available!).

Garden Grove, CA
G.F.R.N. Data Exchange RBBS, (714) 534-1547, Doug Laing
24 hour operation
300 and 1200 baud
5 Mb of files on 4 drives
Special interest in amateur radio and Apple/CPM. This is the second G.F.R.N. system.

Larkspur, CA
Larkspur RBBS/RCPM (415) 461-7726, Jim C.
24 hour operation
Up as of 3/20/81
110, 300, 450, 710 baud; SPRINT, ITT, MCI
2+ Mb on 2 drives
The system carries general and new CP/M software. System now running MP/M and plans are afoot to install a second telephone line, making it the first multi-user remote system. SYSOP will assist others in bringing up MP/M remotely.

Palos Verdes, CA
G.F.R.N. Data Exchange RBBS, (213) 541-2503, Skip Hansen
24 hour operation
300 and 1200 baud; SPRINT, MCI, ITT
2.4 Mb of files on 2 drives
Standard CP/M software with special interest in ham radio-related programs. Soon (with MP/M) will also be reachable through 450 MHz radio.

San Diego, CA
San Diego RCPM, (714) 271-5615, Brian Kantor
No call back; 24 hour operation
300 and 1200 baud; ITT, SPRINT, MCI
2.4 Mb of files on 2 drives
Now with 212-compatible modem; yet another 1200-baud.

East Central

Allentown, PA
Allentown RBBS/RCPM System, (215) 398-3937, Bill Ernest
No call back; 24 hour operation
110, 300, 450, 600, 710 baud; SPRINT and ITT
4.25 Mb of files on hard disk (=4 logical disks)
General CP/M software. Bulletin board of the Lehigh Valley Computer Club.

Grafton, VA
Grafton, VA RBBS, (804) 898-7493, Dave Holmes
No call back, no A.L.D.S; 24 hour operation
300 baud
200 Kb of files on 2 drives
Carries CP/M, TRS-80 and Apple software; plans for setting up a dual system (on one line) with an LNW-80 as well as the CP/M computer. Active as bulletin board.

McClean, VA
RLP RCP/M, McClean, VA, (703) 524-2549, Bob Plouffe
No call back; 24 hour operation
SPRINT and MCI
4 (N*) drives with 640 Kb of files
Running CBBS for messages. New system.

Midwest

Chicago (area), IL
Calamity Cliffs Computer Center, (312) 234-9257
No call back; 1400-0200 daily
300, 450, 600 baud; ITT, SPRINT, MCI
11 Mb of files on a hard disk and 2 floppies
Many of the CPMUG and SIG/M programs available by request.

Chicago, IL
HUG-CBBS, (312) 671-4992, Paul Mayer, Dave Leonard
No call back; 2300 to 1900, 7 days/week
300 baud; SPRINT, ITT, MCI
2 Mb of files on 2 drives
H89-based, operated for the Heath-Zenith Users' Group and with a special interest in H19- and H89-adapted (as well as general CP/M) software.

Hyde Park, IL
Hyde Park RCPM/RBBS, (312) 955-4493, Ben Bronson
No call back; 0100-1700 daily
110, 300, 450, 600, 710 baud; SPRINT, ITT, MCI
2 Mb of files on 2 drives
Special interest in hardware and software reviews, C programs, and very recent releases of standard programs. SYSOP now testing substantial upgrade of RBBS programs.

Royal Oak, MI
Royal Oak CP/M, (313) 759-6569, Keith Petersen

Call back; 24 hour operation
110, 300, 450, 600 baud – 1200 baud modem now available on request; use CHAT or leave a message if you want the 212A switched in – ITT, SPRINT, MCI
600 Kb on two floppy drives and 10 Mb on hard disk (=2 logical drives)
Emphasis on new programs and recent updates of standard programs.

Westland, MI
Westland, MI RBBS/RCPM, (313) 729-1905, Ron Fowler
Call back; 24 hour operation
110, 300, 450 and 600 baud; SPRINT, MCI, ITT
1.4 Mb of files on 2 drives
Emphasis on very recent releases.

Northeast

Lexington, MA
Superbrain RCPM, (617) 862-0781, Paul Kelly
1900-0700 Weekdays, 24 hours weekends
110, 300, 1200 baud; SPRINT, ITT, MCI
300 Kb files on-line
Special interest in Superbrain-adapted CP/M programs.

Bearsville, NY
Bearsville Town SJBBS, (914) 679-6559, Hank Szyszka
No call back, no A.L.D.S.
110, 300, 450, 600, 710 baud
2 Mb of files on 4 drives
Installing MP/M. All CP/MUG programs available by request. General CP/M software.

Rochester, NY
Rochester RBBS, (716) 223-1100, Arnie McGall
No call back; 24 hour operation
110 and 300 baud; SPRINT, MCI, ITT
1.8 Mb of files on 3 drives
S-100 based. General CP/M software. The standard RBBS/RCPM system co-exists with a separate passworded message system called DataStar, which can be entered from CP/M but runs on a separate computer. 600 baud capability expected soon.

Hamilton, ON
Hamilton Area Packet Radio Network (HAPN), (416) 335-6620, Stu Beal
No A.L.D.S; 24 hour operation
110, 300, 450, 600, 710 baud
@? Kb files on-line
New system. System is also linked to radio network and may be accessed via ham radio. Special interest in radio software.

Toronto, ON
Mississauga, Ontario RCPM, (416) 826-5394, Jud Newell
No A.L.D.S.; 24 hour operation
110, 300, 450, 600, 710, 1200 baud
20 Mb hard disk now on-line 24 hours a day
1200 baud Vadic/Bell 212A standard both supported. 300/1200 baud modem available Monday-Friday, PMMI weekends. XMODEM, DISKMENU.DOC and MAST.CAT for details of over 3000 available files. System now restricted to CP/M users only by use of a familiarity question.

Toronto, ON
Mississauga, Ontario HUG-RCP/M, (416) 273-3011
No A.L.D.S.; 1800-0600 weekdays, 24 hours weekends
110, 300, 450, 600, 710 baud
2+ Mb of files on 5 drives
Toronto Heath Users' Group.

South

Huntsville, AL
NACS/UAH RBBS/RCPM, (205) 895-6749, Don Wilkes
Call back, no A.L.D.S.; 24 hour operation
110, 300, 450, 600 baud
700 Kb files on 4 drives
Run for North Alabama Computer Society at University of Alabama; general CP/M software.

Louisville, KY
Louisville RBBS/RCPM, (502) 245-7811, Mike Jung
No call back; 0900-2100 weekdays, 24 hours weekends
300 baud; SPRINT, MCI
2.5 Mb of files on 5 drives
Heath/Zenith-based. Emphasis on BASIC software. Some HDOS stuff available for downloading.

Fort Mill, SC
Fort Mill RIBBS, (803) 547-6576, Bill Taylor
Up as of 08/15/81
No call back, no A.L.D.S.; 24 hour operation
300 and 1200 baud
3 Mb of files on 2 drives
Heath/Zenith-based with 212-compatible modem. The system carries ham stuff, general software, and on-line games. The fourth 1200-baud RCM.

(Listing begins at right)

MBOOT.ASM

(Text begins on page 42)

```
;          Received from Kelly Smith's CP/M-NET (tm)
;          For publication in Doctor Dobb's Journal
;
;          ASSEMBLED, LOADED AND EXECUTED AS SHOWN
;               (except for custom equates)
;
; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ;

; MBOOT.ASM ver 3.0
; (revised 5 Nov 80)
;
;MODEM BOOT PROGRAM by Keith Petersen, W8SDZ.
;Thanks to John Taylor for idea of incorporating
;simple terminal routine.  Based on MODEM.ASM V2.0,
;by Ward Christensen with enhancements from MODEM527.
;
;CP/M - CP/M file transfer program (receive-only)
;
;This program is intended for use as a way to
;initially transfer MODEM.COM or MODEM.ASM to
;a new user.  It may be used with almost any
;modem (see equates).  If PMMI or D.C. Hayes modem
;is used, then uses 'ORIGINATE' mode, 300 baud.
;
;A simple terminal routine at start of program allows
;user to communicate with a remote system prior to
;receiving a file to make it possible to down-load
;a file without intervention on the part of the host
;system's operator.
;
;COMMANDS:  MBOOT FILENAME.FILETYPE
; or  MBOOT A:FILENAME.FILETYPE
; or  MBOOT B:FILENAME.FILETYPE
;
;The program will operate as a dumb terminal until
;an 'ESC' key is typed (ctrl-[).  It then branches
;to the receive routine.  The user may also exit
;to CP/M without opening the receive file by typing
;ctrl-E from the terminal.  The values for the escape
;and exit keys may be changed in accordance with the
;needs of the user - some keyboards do not have the
;'ESC' key and/or provision for control characters.
;See equates.
;
;NOTE: Comments for the source code and tabs have been
;removed to make this file easier to transport from one
;system to another.  (KBP)
;
;NOTE:  NOTE:   TABS have been re-expanded for DDJ format
;
FALSE   EQU     0
TRUE    EQU     NOT FALSE
;
STDCPM  EQU     TRUE    ;TRUE, IS STANDARD CP/M
ALTCPM  EQU     FALSE   ;TRUE, IS ALTERNATE CP/M FOR H8 OR TRS80
;
        IF      STDCPM
BASE    EQU     0
        ENDIF
;
        IF      ALTCPM
BASE    EQU     4200H
        ENDIF
;
EXITCHR EQU     05H     ;CTL-E TO EXIT TERM MODE TO CP/M
ESC     EQU     1BH     ;ESCAPE TO EXIT TERM MODE TO FILE RCVE
;
FASTCLK EQU     FALSE   ;TRUE FOR 4 MHZ CLOCK
;
PMMI    EQU     TRUE    ;TRUE, IS PMMI MODEM
DCH     EQU     FALSE   ;TRUE, IS D.C. HAYES MODEM
;
;IF YOU ARE USING AN EXTERNAL MODEM (NOT S-100 PLUG-IN)
;CHANGE THESE EQUATES FOR YOUR MODEM PORT REQUIREMENTS
;
INITREQ EQU     FALSE   ;TRUE, IF MODEM PORT INIT. REQ'D
INITC1  EQU     0AAH    ;1ST INIT CHAR TO CTL PORT FOR USART
INITC2  EQU     40H     ;2ND  "
INITC3  EQU     4EH     ;3RD  "
INITC4  EQU     37H     ;4TH  "
;
        IF NOT PMMI AND NOT DCH
MODCTLP EQU     02H     ;PUT YOUR MODEM CONTROL PORT HERE
MODSNDB EQU     80H     ;YOUR BIT TO TEST FOR SEND
MODSNDR EQU     80H     ;YOUR VALUE WHEN READY TO SEND
MODRCVB EQU     40H     ;YOUR BIT TO TEST FOR RECEIVE
MODRCVR EQU     40H     ;YOUR VALUE WHEN READY TO RECEIVE
MODDATP EQU     03H     ;YOUR MODEM DATA PORT
        ENDIF           ;END OF EXTERNAL MODEM EQUATES
;
        IF PMMI
MODCTLP EQU     0C0H
MODSNDB EQU     1
MODSNDR EQU     1
MODRCVB EQU     2
MODRCVR EQU     2
MODDATP EQU     0C1H
BAUDRP  EQU     0C2H
MODCTL2 EQU     0C3H
ORIGMOD EQU     1DH
ANSWMOD EQU     1EH
        ENDIF
;
        IF DCH
MODCTLP EQU     82H
MODSNDB EQU     2
MODSNDR EQU     2
MODRCVB EQU     1
MODRCVR EQU     1
MODDATP EQU     80H
MODCTL2 EQU     81H
ORIGMOD EQU     87H
ANSWMOD EQU     83H
        ENDIF
;
ERRLIM EQU      10
;
SOH     EQU     1
EOT     EQU     4
ACK     EQU     6
NAK     EQU     15H
CAN     EQU     18H
LF      EQU     10
CR      EQU     13
;
BDOS    EQU     BASE+5
```

(Continued on column 2)

(Continued on next page)

MBOOT.ASM

(Listing continued, text begins on page 42)

```
FCB     EQU     BASE+5CH
;
        ORG     BASE+100H
;
        LXI     H,0
        DAD     SP
        SHLD    STACK
        LXI     SP,STACK
        CALL    INITADR
        CALL    ILPRT
        DB      'MBOOT as of '
        DB      '11/5/80',CR,LF,0
        LDA     FCB+1
        CPI     ' '
        JNZ     TERMI
        CALL    ILPRT
        DB      '++NO FILE NAME SPECIFIED++',CR,LF,0
        JMP     EXIT
;
TERMI   CALL    INITMOD
        IN      MODDATP
        IN      MODDATP
        CALL    ILPRT
        DB      CR,LF,'TERMINAL MODE',CR,LF
        DB      'CTL-E exits to CP/M, ESC starts file transfer'
        DB      CR,LF,0
;
TERM    CALL    STAT
        JZ      TERML
        CALL    KEYIN
        CPI     EXITCHR
        JZ      EXIT
        CPI     ESC
        JZ      RCVFIL
        OUT     MODDATP
;
        IF NOT DCH
TERML   IN      MODCTLP
        ENDIF
;
        IF DCH
TERML   IN      MODCTL2
        ENDIF
;
        ANI     MODRCVB
        CPI     MODRCVR
        JNZ     TERM
        IN      MODDATP
        ANI     7FH
        CALL    TYPE
        JMP     TERM
;
RCVFIL  CALL    ERASFIL
        CALL    MAKEFIL
        CALL    ILPRT
        DB      'FILE OPEN, READY TO RECEIVE',CR,LF,0
;
RCVLP   CALL    RCVSECT
        JC      RCVEOT
        CALL    WRSECT
```

(Continued on top of page 48)

```
        CALL    INCRSNO
        CALL    SENDACK
        JMP     RCVLP
;
RCVEOT  CALL    WRBLOCK
        CALL    SENDACK
        CALL    CLOSFIL
        CALL    ERXIT
        DB      CR,LF,'TRANSFER COMPLETE$'
;
RCVSECT XRA     A
        STA     ERRCT
;
RCVRPT  MVI     B,10
        CALL    RECV
        JC      RCVSERR
        CPI     SOH
        JZ      RCVSOH
        ORA     A
        JZ      RCVRPT
        CPI     EOT
        STC
        RZ
;
RCVSERR MVI     B,1
        CALL    RECV
        JNC     RCVSERR
        MVI     A,NAK
        CALL    SEND
        LDA     ERRCT
        INR     A
        STA     ERRCT
        CPI     ERRLIM
        JC      RCVRPT
;
RCVSABT CALL    CLOSFIL
        CALL    ERXIT
        DB      '++UNABLE TO RECEIVE BLOCK'
        DB      CR,LF,'++ABORTING++$'
;
RCVSOH  MVI     B,1
        CALL    RECV
        JC      RCVSERR
        MOV     D,A
        MVI     B,1
        CALL    RECV
        JC      RCVSERR
        CMA
        CMP     D
        JZ      RCVDATA
        JMP     RCVSERR
;
RCVDATA MOV     A,D
        STA     RCVSNO

        MVI     C,0
        LXI     H,BASE+80H
;
RCVCHR  MVI     B,1
        CALL    RECV
        JC      RCVSERR
        MOV     M,A
        INR     L
        JNZ     RCVCHR
        MOV     D,C
        MVI     B,1
        CALL    RECV
```

(Continued on page 49, column 1)

```
        JC      RCVSERR
        CMP     D
        JNZ     RCVSERR
        LDA     RCVSNO
        MOV     B,A
        LDA     SECTNO
        CMP     B
        JZ      RECVACK
        INR     A
        CMP     B
        JNZ     ABORT
        RET
;
RECVACK CALL    SENDACK
        JMP     RCVSECT
;
SENDACK MVI     A,ACK
;
SEND    PUSH    PSW
        ADD     C
        MOV     C,A
;
        IF NOT DCH
SENDW   IN      MODCTLP
        ENDIF
;
        IF DCH
SENDW   IN      MODCTL2
        ENDIF
;
        ANI     MODSNDB
        CPI     MODSNDR
        JNZ     SENDW
        POP     PSW
        OUT     MODDATP
        RET
;
ABORT   LXI     SP,STACK
;
ABORTL  MVI     B,1
        CALL    RECV
        JNC     ABORTL
        MVI     A,CAN
        CALL    SEND
;
ABORTW  MVI     B,1
        CALL    RECV
        JNC     ABORTW

        MVI     A,'
        CALL    SEND
        CALL    ERXIT
        DB      'MBOOT PROGRAM CANCELLED$'
;
INCRSNO LDA     SECTNO
        INR     A
        STA     SECTNO
        RET
;
ERASFIL LXI     D,FCB
        MVI     C,17
        CALL    BDOS
        INR     A
        RZ
        CALL    ILPRT
        DB      '++FILE EXISTS, TYPE Y TO ERASE:',0
```

(Continued on column 2)

```
        CALL    KEYIN
        PUSH    PSW
        CALL    TYPE
        CALL    CRLF
        POP     PSW
        ANI     5FH
        CPI     'Y'
        JNZ     MXIT
        LXI     D,FCB
        MVI     C,19
        JMP     BDOS
;
MAKEFIL LXI     D,FCB
        MVI     C,22
        CALL    BDOS
        INR     A
        RNZ
        CALL    ERXIT
        DB      '++ERROR - CAN''T MAKE FILE',CR,LF
        DB      '++DIRECTORY MUST BE FULL$'
;
CLOSFIL LXI     D,FCB
        MVI     C,16
        CALL    BDOS
        INR     A
        RNZ
        CALL    ERXIT
        DB      '++CAN''T CLOSE FILE$'
;
WRSECT  LHLD    SECPTR
        XCHG
        LXI     H,BASE+80H
        CALL    MOVE128
        XCHG
        SHLD    SECPTR
        LDA     SECINBF
        INR     A
2STA    SECINBF
        CPI     16
        RNZ
;
WRBLOCK LDA     SECINBF
        ORA     A
        RZ
        MOV     C,A
        LXI     D,DBUF
;
DKWRLP  PUSH    H
        PUSH    D
        PUSH    B
        MVI     C,26
        CALL    BDOS
        LXI     D,FCB
        MVI     C,21
        CALL    BDOS
        POP     B
        POP     D
        POP     H
        ORA     A
        JNZ     WRERR
        LXI     H,80H
        DAD     D
        XCHG
        DCR     C
        JNZ     DKWRLP
        XRA     A
        STA     SECINBF
```

(Continued on column 3)

```
        LXI     H,DBUF
        SHLD    SECPTR
;
RSDMA   LXI     D,BASE+80H
        MVI     C,26
        JMP     BDOS
;
WRERR   CALL    RSDMA
        CALL    ILPRT
        DB      '++ERROR WRITING FILE',CR,LF,0
        JMP     ABORT
;
RECV    PUSH    D
;
        IF      FASTCLK
        MOV     A,B
        ADD     A
        MOV     B,A
        ENDIF
;
MSEC    LXI     D,50000
;
        IF NOT DCH
MWTI    IN      MODCTLP
        ENDIF
;
        IF      DCH
MWTI    IN      MODCTL2
        ENDIF
;
        ANI     MODRCVB
        CPI     MODRCVR
        JZ      MCHAR
        DCR     E
        JNZ     MWTI
        DCR     D
        JNZ     MWTI
        DCR     B
        JNZ     MSEC
        POP     D
        STC
        RET
;
MCHAR   IN      MODDATP
        POP     D
        PUSH    PSW
        ADD     C
        MOV     C,A
        POP     PSW
        ORA     A
        RET
;
INITADR LHLD    BASE+1
        LXI     D,3
        DAD     D
        SHLD    VSTAT+1
        DAD     D
        SHLD    VKEYIN+1
        DAD     D
        SHLD    VTYPE+1
        RET
;
INITMOD EQU     $
;
        IF      INITREQ
```

(Continued on next page)

MBOOT.ASM

(Listing continued, text begins on page 42)

```
        MVI     A,INITC1
        OUT     MODCTLP
        NOP ! NOP ! NOP
        NOP ! NOP
        MVI     A,INITC2
        OUT     MODCTLP
        NOP ! NOP ! NOP
        NOP ! NOP
        MVI     A,INITC3
        OUT     MODCTLP
        NOP ! NOP ! NOP
        NOP ! NOP
        MVI     A,INITC4
        OUT     MODCTLP
        NOP ! NOP ! NOP
        NOP ! NOP
        ENDIF
;
        IF      PMMI
        MVI     A,52
        OUT     BAUDRP
        MVI     A,7FH
        OUT     MODCTL2
        ENDIF
;
        IF      DCH
        MVI     A,16H
        OUT     MODCTL2
        ENDIF
;
        IF      PMMI OR DCH
;
        LXI     H,4000
;
OFFDLY  DCR     L
        JNZ     OFFDLY
        DCR     H
        JNZ     OFFDLY
        MVI     A,ORIGMOD
        OUT     MODCTLP
        ENDIF
;
        RET             ;<--THIS RET MUST BE HERE
;
CRLF    MVI     A,CR
        CALL    TYPE
        MVI     A,LF
;
TYPE    PUSH    PSW
        PUSH    B
        PUSH    D
        PUSH    H
        MOV     C,A
VTYPE   CALL    $-$
        POP     H
        POP     D
        POP     B
        POP     PSW
        RET
;
STAT    PUSH    B
        PUSH    D
        PUSH    H
VSTAT   CALL    $-$
        POP     H
        POP     D
        POP     B
        ORA     A
        RET
;
KEYIN   PUSH    B
        PUSH    D
        PUSH    H
VKEYIN  CALL    $-$
        POP     H
        POP     D
        POP     B
        ANI     7FH
        RET
;
ILPRT   XTHL
;
ILPLP   MOV     A,M
        ORA     A
        JZ      ILPRET
        CALL    TYPE
        INX     H
        JMP     ILPLP
;
ILPRET  XTHL
        RET


;
ERXIT   POP     D
        MVI     C,9
        CALL    BDOS
;
MXIT    CALL    ILPRT
        DB      CR,LF,'DON''T FORGET TO DISCONNECT MODEM'
        DB      CR,LF,0
;
EXIT    LHLD    STACK
        SPHL
        RET
;
MOVE128 MVI     B,128
;
MOVE    MOV     A,M
        STAX    D
        INX     H
        INX     D
        DCR     B
        JNZ     MOVE
        RET
;
RCVSNO  DB      0
SECTNO  DB      0
ERRCT   DB      0
EOFLG   DB      0
SECPTR  DW      DBUF
SECINBF DB      0
        DS      60
STACK   DS      2
DBUF    EQU     $
;
        END
```

End Listing

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

Late News for 8087 Fanciers

Intel recently dropped the price of the Intel 8087 numeric processor to $230 in single-unit quantities. Some high-level software tools to exploit the 8087 are also beginning to appear. In addition to my own company's version of Forth which can use the 8087 co-processor, Lifeboat Associates is marketing the Lattice C Compiler which includes an 8087 runtime library, and rumor tells of a UCSD Pascal support file named 8087.PLS which is not yet officially announced.

16-Bit Tools, Installment # 2

The accompanying listing is an 8086/88 assembly language subroutine that returns the sine or cosine of an angle to four significant digits. The precision is adequate for most graphics applications, and the routines are implemented by a combination of deduction and table-lookup techniques that are extremely fast. The original version of these routines was coded in Forth by John James and placed in the public domain; interested readers are referred to his article in *Forth Dimensions*, volume 4, no. 1.

Readers who also need a tangent function can derive it easily:

$$\tan(x) = \frac{\sin(x) * 10000}{\cos(x)}$$

But watch out: finding tangents in this way for angles greater than seventy-two degrees will lead to a divide overflow, which causes a hardware interrupt on the 8086/88.

A more precise (but slower) method of finding sine, cosine and tangent through a simple polynomial expansion will be presented in a later column.

Product Report: Microsoft RAMCard with RAMDrive

This integrated combination of hardware and software can be installed in any IBM Personal Computer, and will markedly enhance its operation. "RAMDrive" refers to a technique of mapping a disk directory and file structure onto a portion of the machine's random access memory. The operating system drivers are modified so that the simulated disk can be accessed by application programs like any other peripheral device. Since there are no moving parts, data can be retrieved nearly instantaneously. As many as three RAMCards may be installed, yielding a maximum of 768 Kbytes of RAMDrive storage.

The package contains a memory expansion board of average-quality construction socketed for 256 Kbytes with parity, a diskette of utility programs and a sixty-page manual. The product is available in several versions and prices, depending on the amount of RAM initially supplied. The user can upgrade a partially filled card by simply plugging in more dynamic memory chips.

Installing the memory card in the IBM PC is quite simple, and can be done in a few minutes by anyone who can read and handle a screwdriver. The manual is clear and complete, and has plenty of helpful troubleshooting tips.

Next, you use a memory test provided on the Microsoft diskette to make sure the RAMCard is working properly. This program is a classic of its kind. It was obviously written by a programmer who was told to make the product "user-friendly," and who thought he could meet that objective with liberal use of reverse-video, blinking text, complicated screen-oriented presentations and, of course, using the special-function keys at any cost. For example, the user is asked to respond to one of the program's setup questions by pushing either the F9 function button or the space bar, neither of which are conveniently located or have the slightest mnemonic value. In spite of its veneer of razzle-dazzle, the program suffers from major design faults in the human interface that make it awkward to use, such as a lack of backspace capability should you be unfortunate enough to make an error in numeric entry. The program and documentation don't provide the smallest morsel of information about the testing process itself, and the only way to terminate the test once it is running is to reset the computer. This is not my idea of a classy program.

After you are convinced that the memory card is working correctly, you run the configuration program, which patches the PC-DOS operating system and makes the extra memory available as a simulated disk device. This is a simple procedure and requires only a few seconds – afterwards, you can copy the revised operating system to your various working disks. You might wonder what would prevent someone from pirating the configuration programs and using them with other companies' less-expensive memory cards. Never fear, Microsoft has provided against this eventuality: part of the necessary software is in a programmed read-only memory chip installed right on the RAMCard.

Using the RAMDrive capability can result in astonishing improvements in performance for programs which are disk I/O bound. For example, a simple Forth program to sequentially read through a 100 Kbyte file, accessing it as 1024-byte records, completed in 24.9 seconds using a regular 5¼-inch floppy disk drive but only required one second on the RAMDrive.

The RAMCard and RAMDrive package costs $495 with 64Kbytes installed, and each additional 64Kbytes costs you $200 from Microsoft up to a grand total of $1095 for a complete 256Kbyte card. Here is where a little careful shopping can save you a bundle of money. I bought the minimum 64Kbyte card from Microsoft via Computerland, and obtained the twenty-seven dynamic RAM (type 4164) chips necessary to fill up the card for $227 from BG Micro in Garland, Texas. Thus, I saved $373 as compared to buying the RAM expansions from Microsoft.

Note that if you do not want the RAMDrive feature of this product, there are many memory cards available from reputable companies at considerably lower cost. But if you really need increased speed from your IBM PC, but aren't yet ready to take the plunge on a hard disk, then this product is great!

(Listing begins at right)

8086 Trig

(Text begins on page 51)

```
                                title   'Trig Lookup Functions'
                                pagewidth       79
                        ;
                        ; Trig lookup functions for 8086/88
                        ; adapted by Ray Duncan from a public
                        ; domain FORTH routine written by John James.
                        ;
                        ;
                        ; 'Trig' is a common routine used by 'sin'
                        ; and 'cos' to reduce an angle into the
                        ; range 0-90 degrees then extract the
                        ; appropriate function value from the table.
                        ;
0000 8BD8               trig:   mov     bx,ax   ;bx <- degrees
0002 83FB5A                     cmp     bx,90   ;if > 90 degrees
0005 7E06       000D            jle     trig1   ;reduce the angle
0007 81EBB400                   sub     bx,180
000B F7DB                       neg     bx
000D D1E3               trig1:  sal     bx,1    ;table index = 2*deg.
                                                ;now extract value
000F 2E8B873F00                 mov     ax,sintbl [bx]
0014 C3                         ret             ;back to caller
                        ;
                        ;  Cosine lookup:  add 90 degrees to
                        ;  argument and use sine lookup.
                        ;  call with:       ax = degrees
                        ;  returns:         ax = cosine * 10000
                        ;  other registers preserved.
                        ;
0015 055A00             cos:    add     ax,90
                        ;
                        ;  Sine lookup:  reduce to angle in
                        ;  range 0-359 degrees, then call
                        ;  'trig' to extract function value
                        ;  from table.
                        ;  call with:     ax = degrees
                        ;  returns:       ax = sine * 10000
                        ;  other registers preserved.
                        ;
0018 52                 sin:    push    dx      ;save registers
0019 53                         push    bx
001A 99                         cwd             ;deg -> double prec.
001B BB6801                     mov     bx,360  ;reduce angle to
001E F7FB                       idiv    bx      ;range 0-359 degrees
0020 8BC2                       mov     ax,dx   ;let ax=remainder
0022 0BC0                       or      ax,ax   ;is angle negative?
0024 7903       0029            jns     sin2    ;no,jump
0026 056801                     add     ax,360  ;yes,make it positive
0029 3DB400             sin2:   cmp     ax,180  ;now reduce angle to
002C 7E0B       0039            jle     sin3    ;range 0-180 degrees
002E 2DB400                     sub     ax,180  ;angle was > 180 deg.
0031 E8CCFF     0000            call    trig    ;look up function value

0034 F7D8                       neg     ax
0036 E90300     003C            jmp     sin4
0039 E8C4FF     0000    sin3:   call    trig    ;angle was <= 180 deg
003C 5B                 sin4:   pop     bx      ;restore registers
003D 5A                         pop     dx
003E C3                         ret             ;back to caller
                        ;
                        ; lookup table for trig functions
                        ;
003F 0000               sintbl  dw      0       ;0 degrees
0041 AF00                       dw      175     ;1
0043 5D01                       dw      349     ;2
0045 0B02                       dw      523     ;3
0047 BA02                       dw      698     ;4
0049 6803                       dw      872     ;5
004B 1504                       dw      1045    ;6
004D C304                       dw      1219    ;7
004F 7005                       dw      1392    ;8
0051 1C06                       dw      1564    ;9
0053 C806                       dw      1736    ;10
0055 7407                       dw      1908    ;11
0057 1F08                       dw      2079    ;12
0059 CA08                       dw      2250    ;13
005B 7309                       dw      2419    ;14
005D 1C0A                       dw      2588    ;15
005F C40A                       dw      2756    ;16
0061 6C0B                       dw      2924    ;17
0063 120C                       dw      3090    ;18
0065 B80C                       dw      3256    ;19
0067 5C0D                       dw      3420    ;20
```

(Continued on next page)

8086 Trig

(Listing continued, text begins on page 51)

```
0069 000E                       dw      3584    ;21
006B A20E                       dw      3746    ;22
006D 430F                       dw      3907    ;23
006F E30F                       dw      4067    ;24
0071 8210                       dw      4226    ;25
0073 2011                       dw      4384    ;26
0075 BC11                       dw      4540    ;27
0077 5712                       dw      4695    ;28
0079 F012                       dw      4848    ;29
007B 8813                       dw      5000    ;30
007D 1E14                       dw      5150    ;31
007F B314                       dw      5299    ;32
0081 4615                       dw      5446    ;33
0083 D815                       dw      5592    ;34
0085 6816                       dw      5736    ;35
0087 F616                       dw      5878    ;36
0089 8217                       dw      6018    ;37
008B 0D18                       dw      6157    ;38
008D 9518                       dw      6293    ;39
008F 1C19                       dw      6428    ;40
0091 A119                       dw      6561    ;41
0093 231A                       dw      6691    ;42

0095 A41A                       dw      6820    ;43
0097 231B                       dw      6947    ;44
0099 9F1B                       dw      7071    ;45
009B 191C                       dw      7193    ;46
009D 921C                       dw      7314    ;47
009F 071D                       dw      7431    ;48
00A1 7B1D                       dw      7547    ;49
00A3 EC1D                       dw      7660    ;50
00A5 5B1E                       dw      7771    ;51
00A7 C81E                       dw      7880    ;52
00A9 321F                       dw      7986    ;53
00AB 9A1F                       dw      8090    ;54
00AD 0020                       dw      8192    ;55
00AF 6220                       dw      8290    ;56
00B1 C320                       dw      8387    ;57
00B3 2021                       dw      8480    ;58
00B5 7C21                       dw      8572    ;59
00B7 D421                       dw      8660    ;60
00B9 2A22                       dw      8746    ;61
00BB 7D22                       dw      8829    ;62
00BD CE22                       dw      8910    ;63
00BF 1C23                       dw      8988    ;64
00C1 6723                       dw      9063    ;65
00C3 AF23                       dw      9135    ;66
00C5 F523                       dw      9205    ;67
00C7 3824                       dw      9272    ;68
00C9 7824                       dw      9336    ;69
00CB B524                       dw      9397    ;70
00CD EF24                       dw      9455    ;71
00CF 2725                       dw      9511    ;72
00D1 5B25                       dw      9563    ;73
00D3 8D25                       dw      9613    ;74
00D5 BB25                       dw      9659    ;75
00D7 E725                       dw      9703    ;76
00D9 1026                       dw      9744    ;77
00DB 3526                       dw      9781    ;78
00DD 5826                       dw      9816    ;79
00DF 7826                       dw      9848    ;80
00E1 9526                       dw      9877    ;81
00E3 AF26                       dw      9903    ;82
00E5 C526                       dw      9925    ;83
00E7 D926                       dw      9945    ;84
00E9 EA26                       dw      9962    ;85
00EB F826                       dw      9976    ;86
00ED 0227                       dw      9986    ;87
00EF 0A27                       dw      9994    ;88
00F1 0E27                       dw      9998    ;89
00F3 1027                       dw      10000   ;90 degrees

                                end
```

End Listing

OF INTEREST

by Michael Wiesenberg

A Potpourri of Good-Priced Hardware

The three-inch Winchesters are coming! One of the first is from SyQuest Technology. Their **SQ306** they claim is "the industry's first 3.9-inch (100mm) removable cartridge Winchester disk drive." SyQuest is aiming at the OEM market, but the price of $400 per unit in large quantities means we can expect to see these five-megabyte Winchesters arrive reasonably priced. The cartridges will sell in quantity for $30 each. The small size of this unit (1.625 inches tall) allows two SyQuest drives to fit into the same space as one 5.25-inch Winchester or mini-floppy drive. We should see these units integrated, probably stacked two at a time, into the next generation of portable computers. **Reader Service No. 306.**

Monitor the status of the "seven most important" (we are told) RS-232 lines with, what else?, **RS-232 Tester** from B & B Electronics. A female 25-pin connector at one end and a male connector at the other let you attach the Tester directly to the interface and leave it permanently in the line, with LEDs constantly displaying status, but not interfering with data transfer ability. $39.95 postpaid. **Reader Service No. 316.**

The **SSB-MPF Speech Synthesizer Board** from Etronix provides a 400-word vocabulary for the MPF-1 Micro-Professor, the Z-80-based single-board computer-in-a-book described in a previous column. The board does its input/output with the Micro-Professor's keyboard and speaker, and features a 4K time clock and speech utility EPROM, two EPROM sockets that can be used to expand the vocabulary, adjustable pitch and volume controls, a power adaptor, all necessary connection accessories and a manual. $129. **Reader Service No. 326.**

A Gallimaufry of Inexpensive Software

AUTODIFF is a file difference detector for CP/M by the Software Toolworks. Not only does this comparison utility list differences between files or produce a copy of the file with all changes flagged, it also reports insertions, deletions and changes, listing them in ASCII or hexadecimal format to a terminal, printer or disk file. Filters can be set to ignore non-printing characters or to display control characters. AUTODIFF can be used on most files and costs $29.95 (plus $2 postage and handling) on 5-inch disk for Osborne I and Heath/Zenith, and 8-inch CP/M disk. **Reader Service No. 346.**

Disk-Edit is a screen-oriented disk editor from Supersoft for CP/M programs. It can call up files that, according to Supersoft, "are not even accessible with a normal text editor, then edit those files in either ASCII or hexadecimal notation." The program loads a one-kilobyte section of a disk (hard or floppy) into a buffer, and then displays a dual-view "window" into the buffer. On the left are the hexadecimal values of each byte; on the right, the ASCII. Change either, and the corresponding value changes instantly in the other. Simple commands move the cursor up, down, left, right, to the next page or screen, to the start of the file and so on. There are also string searches and many other functions. Disk-Edit has a "terminal definition package" and can be configured to most CP/M systems. $100, or $15 for the manual only. **Reader Service No. 356.**

You probably thought that a text editor for the IBM Personal Computer would be expensive, but **WINDOW**, a full-screen text editor program from Intellect Associates Inc., costs $150. It uses all the screen and keyboard capabilities of the PC, including single-stroke editing commands with the function keys. It's not quite a word processor, but WINDOW does most of the things a word processor can, such as easily move the cursor, scroll in four directions, do global search and replace, insert and delete characters and lines, move, copy, split and join lines and edit text files larger than available memory. WINDOW runs under IBM-DOS, comes on 5.25-inch diskette, with documentation, and requires 64K, one drive, and a monochrome display and adapter. To turn WINDOW into a real word processor, add Intellect's **PCTEXT** for $100, and get a text processing package that indents, centers, controls line spacing and margins, justifies, inserts headings and footings, numbers pages, underlines and merges documents. It, too, runs under IBM-DOS, requires 48K, one drive and a printer. And while we're talking about Intellect Associates, they offer for the same configuration a data management system, **DMS**, a self-prompting, menu-driven system with which users create data entry forms on the screen in user-specified formats, and then enter, retrieve, modify, correct and delete data with the forms and print reports. The sequential ASCII data base files can be accessed by any language. DMS is written in **C88** (Intellect Associates' one-pass-compiler subset of C that costs $250, generates compact 8088 machine code and comes with a linker), which makes it run, they say, very fast, and also makes good use of the PC's screen and keyboard features. It requires 48K, one drive (two recommended) and monochrome display adapter. **Reader Service No. 366.**

Cross-assemblers can cost a bundle, but here's one for the Zilog Z8 that costs $150. **SYSTEM-Z8**, by Allen Ashley, for CP/M, includes a down-loader for the Zilog Z8 Development Module to transmit developed programs for in-circuit test. You get a macro-assembler with macro and conditional assembly and chaining, interactive editor/assembler, text editor, cross-reference generator, complete documentation and user support by mail or phone. Current SYSTEM-Z8 owners can phone for a free update. **Reader Service No. 376.**

DDJ

Letters *(Continued from page 6)*

by *DDJ* readers. First, however, I attacked its primary problem: it's *slow*.

I studied the code for a while, and decided to change the global symbol table from a linearly searched list to a hash table. I also put the macros in the global symbol table, with pointers into the array of macro definitions. Since the original code was well-structured, changing algorithms was not difficult.

I changed "numglbs" from 300 to 512 (a power of two), "SYMTBSZ" from 5040 to 8008, and introduced "#define MASKGLBS 511". I introduced "#define MACRO 5", giving a new possible entry for "ident". Since I'm putting the macro names into the global symbol table rather than the macro table, I changed "macqsize" from 1000 to 500. "macptr" has to be initialized to 1 instead of 0. (Initializing it to 0 causes a subtle bug affecting only the first macro. I'll leave the details as an exercise for the student.)

I added the following initializing code to main( ):

```
        glbptr=STARTGLB;
        while(glbptr<ENDGLB){
                *glbptr=0;
                glbptr=glbptr+SYMSIZ;
        }
        glbptr=STARTGLB+SYMSIZ*5;
                /* clear global symbols
*/
```

"dumpglbs( )" needed the declaration "int i;", and the top of the loop changed to

```
    i=NUMGLBS;
    while(i--){  /* 6/19/82  jrvz */
        if(*cptr){
            if((cptr[ident]!=function)
            &(cptr[ident]!=MACRO))
            /* do if anything but
            function or macro
            jrvz 6/19/82 */
```

I introduced a simple hash function:

```
hash(sname)
    char *sname;
{   int h,c;
    h=*sname;
    while(c=*(++sname)) h=(h<<1)+c;
    return h;
}
```

I then rewrote the following routines where boxes indicate changes):

```
findglb(sname)
    /* cptr is set to entry if found,
    or appropriate empty slot if not */
    char *sname;
{   int h;
    h=hash(sname)&MASKGLBS;
    cptr=STARTGLB+h*SYMSIZ;
    while(0==
    astreq(sname,cptr,namemax)){
        if(*cptr==0) return 0;
        cptr=cptr+SYMSIZ;
        if(cptr==ENDGLB)cptr=STARTGLB;
    }
    return cptr;
}


addglb(sname,id,typ,value)
    char *sname,id,typ;
    int value;
{   char *ptr;
    if(findglb(sname))return cptr;
    if(glbptr>=ENDGLB)
        { error("global symbol table
overflow");
        return 0;
        }
    ptr=cptr;
    while(an(*ptr++ = *sname++));
            /* copy name */
    cptr[ident]=id;
    cptr[type]=typ;
    cptr[storage]=statik;
    cptr[offset]=value;
    cptr[offset+1]=value>>8;
    glbptr=glbptr+SYMSIZ;
    return cptr;
}


addmac()
{   char sname[namesize];

    if(symname(sname)==0)
        {illname();
        kill();
        return;
        }
    addglb(sname,MACRO,0,macptr);
    while(ch()==' ' | ch()==9)
        gch();
    while(putmac(gch()));
    if(macptr>=macmax)
        error("macro table full");
    }


findmac(sname)
        /* rewritten 6/19/82  jrvz */
    char *sname;
{   if((findglb(sname)!=0)&
    (cptr[ident]==MACRO))
        {return((cptr[offset]&255)+
        (cptr[offset+1]<<8));
        }
    return 0;
}
```

As a result of these changes, the compiler speed has more than doubled. I also enlarged the disk buffers from 128 bytes to 512 bytes. That sped compilation up a further 20%, to about 220 lines/minute on my 2.5 MHz Z-80.

Has anyone installed floating-point variables? Does anyone have a C interpreter, that would let me debug my code quickly? (Note that Tiny-c accepts a different language.)

Keep the Small-C articles coming!

Yours,
James R. Van Zandt
26 Shelton St.
Nashua, NH 03062

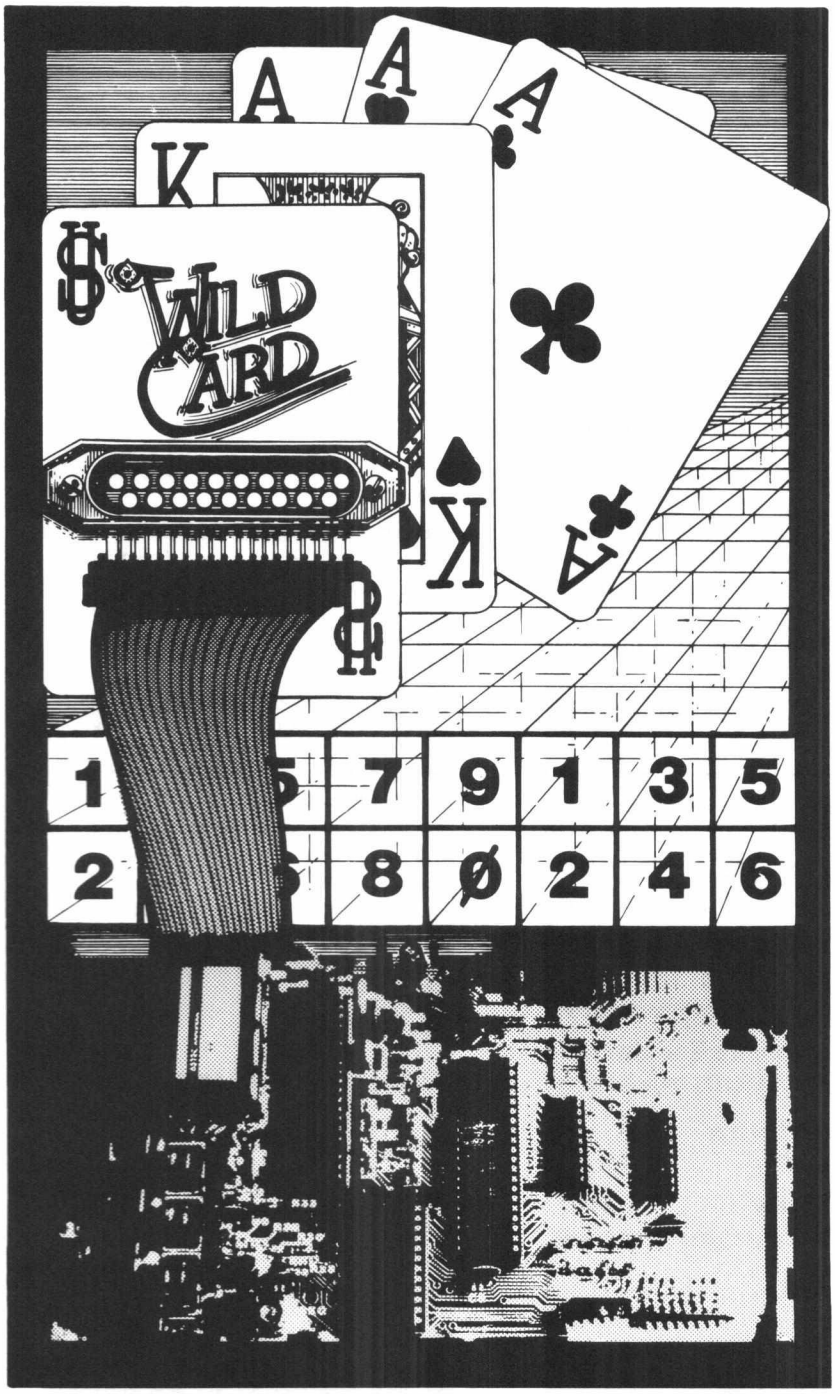
Editor's Note: Readers who are following the developments in Small-C, which originated with Ron Cain's Small-C compiler, should keep a close eye on DDJ *in the coming months. We have a number of exciting articles and listings scheduled.*

DDJ

#73 November 1982 $2.50 ($3.00 in Canada)

Dr. Dobb's Journal

For Users of Small Computer Systems



UNIX Wildcard Filenames

IBM-PC Subroutines

Hash 'n Pascal

Novice Builds Sinclair's ZX81

Memory Management in BASIC

and more inside!

plus: CP/M-86 vs. MSDOS Benchmarks

Dr. Dobb's Journal
For Users of Small Computer Systems

November 1982 Volume 7, Issue 11

Editor – Marlin Ouverson
Assistant Editor – Fritzi Lareau
Contributing Editors –
Dave Caulkins, Dave Cortesi,
Ray Duncan, Gene Head,
Michael Wiesenberg
Marketing Director – Craig Harper
Marketing Assistant – Beatrice Blatteis
Advertising Director – Carl Landau
Circulation Manager – Gloria Romanoff
Circulation Assistants –
Terri Marty, Linda Marohn
Art Director – Clifford West
Production Manager – Barbara Ruzgerian
Production Assistant – Jane A. McKean
Typesetter – Renny Wiggins
Cover Illustration – Al McCahon

© 1982 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Thomas H. Grohne, G. V. Elkins, Robert M. Connors, Robert C. Luckey, Robert N. Stabler, Dewitt S. Brown, Burks A. Smith, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Howard E. Decker, Jim Wilson, Frank Lawyer, Chris Pettus, Paul Lutus, Ben Goldfarb, Rodney Black, Steven Weiss, Pat Phelan, John Brodie, James Flemming, Unison World, Thomas Davis, Jan Steinman, Ronald E. Johnson, G. Hunter, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor, Jr. **Sustaining Subscribers:** Pete Roberts, Steven Fisher Enterprizes. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Hofacker-Verlag, Tegernseer Strass 18, D-8150 Holzkirchen, West Germany. **Asia & Australia:** ASCII Publishing Co. Ltd, 4F Segawa Building 5-2-2, Jingumae, Shibuya-ku, Tokyo 150, Japan.

CONTENTS

Articles

12 Expand Wildcards Under UNIX
by Anthony Skjellum
Unix users know their system provides for wildcard filename expansion at execution. Along with this article, you will find code to enable wildcard filename expansion from within a program, increasing the flexibility and available power.

18 Tests for Pidgin
by William A. Gale
Ever since programmer Gale published Pidgin in *DDJ*, our readers have been praising its power, portability and economical size. This month its creator provides seven test programs, essentially a functional definition of the language, to help implementors catch any bugs in their own versions.

28 A Novice Builds the Sinclair ZX81
by John Connell, Jr.
When he began, this author couldn't tell an IC from PCB, and had never soldered in his life. The perfect choice, we thought, for a test of Sinclair's kit version of the ZX81. With the retail price of the pre-assembled computer at $100, why bother building one? Read the author's account and judge for yourself.

32 68000 Cross Assembler
by Allen Kossow
This listing completes the presentation of Kossow's cross assembler (the first installment was published in our last issue). It was created to run on the author's multi-68000 homebrew system featured in *DDJ* #68.

44 Memory Management in BASIC
by Henry Seymour
Dynamic memory allocation can use array-switching techniques to store more variables than allowed for by RAM size. It's all in how you manage your memory. . .

Departments

6 CP/M Exchange
How to add more BDOS calls; executing machine-language routines under MBASIC; the RCP/M Exchange system.

9 Dr. Dobb's Clinic
The perfect hash, with code for Pascal/Z and CP/M.

46 16-Bit Software Toolbox
CP/M-86 vs. MSDOS benchmarks; Intel 8087 benchmarks; 8086 SQRT.

48 Software Review
"JRT Pascal"

50 Book Review
"Teaching Your Computer to Talk"

52 Of Interest

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, Box E, 1263 El Camino Real, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025 · 415/323-3111

CP/M EXCHANGE

by Gene Head

I said in the very first installment of the CP/M Exchange that you people, the readers, would become the real writers of this column. This month I present two excellent contributions from readers.

Oscar Goldman developed a method of extending the number of system functions available to CP/M programmers. You remember system functions? Set up the C register with the function number, load appropriate data into other registers and then call address 5. My version of CP/M 2.2 supports up through function 40 (decimal). Goldman's program is far better and the additional utilities will be most appreciated by serious programmers. If you have the source code for your BIOS and want to add special functions, give this scheme a try. These additional functions will *not* be universally transportable. However, for a group or club working on modules of a single project, it could enhance the coding process. I'm working on modem communications programs and plan to use this program to add modem functions similar to the console functions (IN, OUT, and STATUS). I think of these additional functions as "on-line macros." The source code with some preliminary comments from Mr. Goldman begins on page 7. Don't expect to find this caliber of program in any other magazine!

Cal Sondgeroth has written a fifty-line program in MBASIC that will load a machine language program into memory for execution under MBASIC. I imagine just about any BASIC program could be made more efficient by writing some of the code in assembler, then use Cal's loading program to load the machine code. This self-documenting program (Listing 2) begins on page 7.

I have established a Remote CP/M System for the CP/M Exchange. The primary purpose of the *CP/M Exchange* RCP/M is to receive your material for this column and provide the programs you will need to establish your own RCP/M, as well as the most current release of Christensen's MODEM program. The *CP/M Exchange* RCP/M is tutorial, and extremely friendly to first-time users. Column material and messages to me should be sent to drive A with file names like YOURNAME.MSG.

The system is Ring-Back. That means the computer sometimes shares the phone with people. You call and let the phone ring once and hang up before the second ring. Call right back and let it ring until the computer answers. The computer is available to callers from 9 p.m. until 9 a.m. (Pacific time) on weekdays, and on weekends by appointment. The Ring-Back computer line is (503) 758-8408. The people phone is (503) 758-0279 from 9 a.m. to 9 p.m. every day. **I cannot return long distance calls, so please don't leave a message for me to call you!**

Finally, remember to register your known-working modem configuration with the CP/M Exchange either by mail or the *CP/M Exchange* RCP/M. That way, others with the same hardware can build on your experiences. When I get a request for help from someone with a similar system, I'll send the request along to you. You as Helper can then contact the Helpee.

Now, get a modem and *go off hook!*

Dear Gene,

This scheme is based on intercepting calls to the BDOS. The only visible entry point into the BDOS is the jump in-

Editorial

Functioning Randomly

The British Broadcasting Corporation isn't sitting still. Not only are they producing a series of microcomputer shows for broadcast, they have arranged the manufacture of their own machine. Viewers who follow the series will be able to "compute along," making television an active learning process. Now, if they only would add interactive cable features. . . One of BBC's people called the other day to let us know they are using a *DDJ* version of Lisp for a program demonstrating non-procedural languages. When I told Doc, he said, "Veni, vidi, vici," but I reminded him that was on the other side of the Channel.

* * *

A short continuation of last month's exhortation on the public domain: No one wants to see his "good will" programming effort become the key part of a piece of big-bucks software with nothing going to the author. So, in the interest of fairness, quite a few authors offer their work freely for individual, non-profit use, but reserve commercial rights. Astute readers know we have been publishing such material for some time – our foremost concern is that great software be available at a reasonable price (the most reasonable price is free, of course, author willing).

But the punch line is that ideas are not copyright-able, only their implementation in code. If someone has a remarkable programming idea, like an electronic spreadsheet calculator, it will find its way around no matter how well-protected the original program. No one has a monopoly on ideas, the only trade currency *DDJ* accepts.

* * *

Speaking of ideas, have you had any lately? We know from the reader surveys many of you received that *DDJ* readers have some interest in advanced utilities for the 6809, and a few other hardware odds and ends. We only repeat ourselves by saying that we can't print what you don't send us. Get the ball rolling and watch the resulting systems work in our pages.

– Marlin Ouverson

struction at the absolute memory address of 5. While it might seem a simple matter to change the address to 6 and 7 in order to effect the interception, many applications programs (e.g., DDT) determine memory size from the information contained there. Therefore, it would seem prudent to leave the content of 6 and 7 alone. Fortunately, for our purpose, the instruction jumped to from 5 is also a jump instruction. This observation is the key to the interception process.

Here is an outline of the modification to be made in the BIOS. (If necessary, change the identifiers to avoid duplication.)

Add the following two equates:

```
MAXFUN    EQU  49    ; Last of the standard BDOS function
                     ;  numbers.
EXTRA     EQU  ?     ; Replace the "?-mark" by the actual
                     ;  number of new BDOS calls.  The new
                     ;   function numbers will start from 50.
```

Toward the end of the warm boot there is a routine called "gocpm". (This name is used by Digital Research in the listing of the skeletal CBIOS.) Somewhere in there will be found the two lines:

```
LXI  H,BDOS
SHLD 6
```

Immediately after these, insert the following 4 lines:

```
LHLD BDOS+1
SHLD NORMAL+1
LXI  H,DIVERT
SHLD BDOS+1
```

and finish with the rest of "gocpm".

At a convenient point in the BIOS add the following:

Expanded BDOS
Listing One

```
NORMAL:   JMP  0      ; The 0 will be replaced during
                      ;  the warm boot.
DIVERT:
     ; This point is reached with the function
     ;  number in C, and the BDOS call information in DE.
     MVI  A,MAXFUN
     CMP  C           ; Is it an old or a new call?
     JNC  NORMAL      ; Old ones are diverted to NORMAL.
     ADI  EXTRA
     CMP  C           ; Are we exceeding the number of
                      ;  allowed calls?
     JNC  OKAY        ; Continue if not.
     LXI  H,0         ; Protocol requires these registers to
     MOV  A,L         ;  be zeroed for out-of-range calls.
     RET
;
OKAY:
     ; Here we handle the new functions.
     LXI  H,0         ; HL is lost.
     DAD  SP
     SHLD SAVSTK      ; Save stack pointer for a later return,
     LXI  SP,SAVSTK   ;  and set up a local stack.
     XCHG
     SHLD NTRPAR      ; Save the entry parameters.
     LXI  D,FIN
     PUSH D           ; A place to return to when done.
     MOV  A,C
     SBI  MAXFUN+1    ; Start counting from 0.
     ADD  A           ; Double it.
     MOV  E,A
     MVI  D,0
     LXI  H,TABLE
     DAD  D           ; HL points to the relevant table entry.
     MOV  E,M         ; Get the address
     INX  H           ;   contained there,
     MOV  D,M         ;     and move it
     XCHG             ;       into (HL).
     PCHL             ; Go do it.
;
FIN:
     ; Return to here when done.
     LHLD SAVSTK
     SPHL             ; Restore old stack pointer.
     LHLD XITPAR      ; Get the result of the routine.
     MOV  A,L
     MOV  B,H         ; As per CP/M protocol.
     RET              ; All done.
;
TABLE:
     ; Jump table for the various new BDOS calls.
     ; There should be one entry for each of "EXTRA" items.
NEW0:     DW   DONEW0
NEW1:     DW   DONEW1
     ...
     ...
;
NTRPAR:   DS   2      ; Save the entry value of DE.
XITPAR:   DS   2      ; Save the result of the BDOS call.
          DS   40     ; Ample local stack -
                      ;  change this if necessary.
SAVSTK:   DS   2      ; Save the stack pointer on entry.
;
;    Each of the following routines must end with a RET instruction.
;         If the routine returns a value, then it should be stored
;              in XITPAR.
;
DONEW0:               ; Actual routine #0.
DONEW1:               ; Actual routine #1.
     etc.
```

Loadhex. Bas
Listing Two

```
10 REM: "LOADHEX.BAS" IS A MICROSOFT BASIC PROGRAM THAT CAN BE USED TO
11 REM: LOAD ASSEMBLY LANGUAGE SUBROUTINES INTO UNUSED MEMORY FROM
12 REM: STANDARD HEX FILES AS OUTPUT BY THE CP/M "ASM" OR "MAC" ASSEMBLERS.
13 REM: ASSEMBLE YOUR SUBROUTINES TO ORG AT THE DESIRED LOAD ADDRESS.
14 REM: THIS CAN BE EITHER IN SOME UNUSED MEMORY OR JUST BELOW THE
15 REM: CP/M BDOS.   YOU CAN LOAD MBASIC WITH REDUCED MEMORY BY USING
16 REM: THE COMMAND "MBASIC /M:&HADRS"  WHERE "ADRS" IS THE HIGHEST
17 REM: ADDRESS THAT MBASIC IS TO USE PUTTING YOUR ASSEMBLER ROUTINES
18 REM: ABOVE THAT POINT
19 REM:
100 CLEAR 1000
110 PRINT CHR$(26)
120 FI$="BASIO.HEX"       :REM NAME OF HEX FILE TO LOAD
130 OPEN "I",#1,FI$
140 PRINT "LOADING ASSEMBLER HEX FILE --> MEMORY"
150 POKE &HD004,1   :REM THIS KEEPS MY DISK HEAD LOADED FOR SLOW I/O
160 INPUT #1,J$:PRINT J$
170 IF MID$(J$,2,2)="00" THEN 490
180 L$=MID$(J$,4,4)             :REM LOAD ADDRESS STRING
190 D$=MID$(J$,10,LEN(J$)-11)  :REM HEX DATA CHAR STRING
200 REM:
210 REM: COMPUTE THE HEX LOAD ADDRESS
220 REM:
230 X=ASC(RIGHT$(L$,1))-48:IF X>9 THEN X=X-7
240 L=X
250 X=ASC(MID$(L$,3,1))-48:IF X>9 THEN X=X-7
260 L=L+(X*16)
270 X=ASC(MID$(L$,2,1))-48:IF X>9 THEN X=X-7
280 L=L+(X*256)
290 X=ASC(LEFT$(L$,1))-48:IF X>9 THEN X=X-7
300 L=L+(X*4096)
310 REM:
320 REM: FETCH ASSEMBLER DATA AND STORE --> MEMORY
330 REM:
340 FOR J=1 TO LEN(D$) STEP 2
350 B$=MID$(D$,J,2)
360 REM:
370 REM: COMPUTE THE 8-BIT VALUE TO PUT IN MEMORY
380 REM:
390 X=ASC(RIGHT$(B$,1))-48:IF X>9 THEN X=X-7
400 B=X
410 X=ASC(LEFT$(B$,1))-48:IF X>9 THEN X=X-7
420 B=B+(X*16)
430 REM:
440 REM: STORE NEXT BYTE AND BUMP LOAD POINTER
450 REM:
460 POKE L,B:L=L+1
470 NEXT J
480 GOTO 160
490 POKE &HD004,0        :REM ALLOWS MY DISK HEAD TO UNLOAD
500 END
```

End Listing

DDJ

DR. DOBB'S CLINIC

by Dave Cortesi

The deflated condition of our mailbag (broad hint) means that this month's Clinic will be both short and long. Short because we don't have much to say; long because the one subject we have is implemented in a fairly sizable program.

Perfect Hashing

Everybody knows what hashing is, right? You have a collection of n unique values, and a table of size m. You also have a function *hash(v)* which, given a value v, returns a index between one and m (or between zero and $m-1$). You store an item v in the table at location TABLE[*hash(v)*]. Normally, the result of *hash(v)* is not unique for all v; some v's yield identical table indexes. The different v's are said to "collide." There are various schemes for handling collisions. When there is a collision, it will take more than one "probe" into the table to find one of the colliding values.

A *perfect* hash function is one that produces no collisions. A perfect hash is a very nice thing to have, because it ensures that you will be able to find any v in just one probe of the table. Perfect hash functions are possible, but they are hard to find. The function that will produce a perfect hash for one set of values will collide all over the place for a different set or for a table of slightly different size. A *minimal* perfect hash is one that maps the n values onto a table of size n, with no extra slots and no collisions.

Algorithms have been published for finding perfect hash functions. A new one appeared recently: "A Letter-Oriented Minimal Perfect Hashing Function" by Curtis R. Cook and R. R. Oldehoeft, in *SigPlan Notices*, Volume 17, No. 9 (Sept. 1982). Although Cook and Oldehoeft did not publish a program, their description of the algorithm seemed clear enough that we, lacking other material for this column (hint, hint), decided to try implementing it ourselves.

Cook's and Oldehoeft's algorithm assumes that the values to be hashed are uppercase keywords in ASCII, and that the function *hash* is defined as

$$hash(A,Z,L,M) = (\text{encoding}[A] + \text{encoding}[Z] + L) \bmod M$$

where A is the initial letter of a keyword, Z is the word's final letter, L is the length of the word, and M is the size of the table. The function depends on a vector of letter codes which might be declared in pseudo-Pascal as

```
var encoding : array['A'..'Z']
    of 0..m-1
```

The aim of the algorithm is to find a permutation of numbers in the "encoding" array which will cause the function *hash* to be minimal and perfect. It is essentially a backtracking algorithm. It proceeds through a list of keywords, looking at their initial and final letters. When it finds a letter that hasn't been assigned an encoding yet, it assigns the next free code. When both numbers have codes, it computes the *hash* function. If that entry of the table is free, it is claimed for that word and the algorithm proceeds. If the table entry isn't free, the encoding to that point isn't perfect. The last-assigned code is given up, and the algorithm backtracks to an earlier point to try another code.

The contribution of Cook and Oldehoeft is in the ordering of the words before the algorithm begins. By ordering the words in a particular way (described in the comments in Listing 1, page 56), they assign the most important codes first. The ordering also allows groups of words to be considered at once. Doing so complicates the program, but speeds it up quite a bit.

There is no way of knowing whether a perfect hash exists for any particular set of keywords. It's entirely possible that the algorithm could run through all possible encodings and report failure. On the other hand, there might be many "encoding" permutations that will serve. The behavior of our program demonstrated this. For some of the sets of keywords displayed by Cook and Oldehoeft, it found solutions in less than a minute. For others, it ran for hours and found nothing. Our program's times seemed to bear no relation to the times reported in the paper that inspired it. We didn't have the patience to let it run to completion in any case – assuming there is time enough in the universe to do so. . . .

While it is running, our program documents its recursive operations on the console. The cursor buzzes in place, surges forward, falls back unexpectedly – it makes quite an amusing movie of a backtracking algorithm in operation.

In an effort (probably a misguided one) to speed up our program, we added

(Continued on page 55)

Letter encodings:

A = 22	H = 9	R = 12
B = 21	I = 11	S = 31
C = 28	L = 1	T = 4
D = 2	M = 5	U = 6
E = 0	N = 10	V = 26
F = 15	O = 7	W = 29
G = 24	P = 17	Y = 13

Hash values:

0 = ARRAY	10 = MOD	20 = RECORD	30 = FOR
1 = VAR	11 = DO	21 = OR	31 = DIV
2 = WITH	12 = UNTIL	22 = REPEAT	32 = CASE
3 = (empty)	13 = TO	23 = IN	33 = FUNCTION
4 = ELSE	14 = NIL	24 = OF	34 = WHILE
5 = END	15 = DOWNTO	25 = PACKED	35 = GOTO
6 = (empty)	16 = OTHERWISE	26 = PROCEDURE	36 = BEGIN
7 = LABEL	17 = NOT	27 = AND	37 = CONST
8 = TYPE	18 = THEN	28 = IF	38 = SET
9 = EXTERNAL	19 = FILE	29 = PROGRAM	39, 40 empty

Table 1

A near-minimal perfect hash function for a set of Pascal keywords.

Expand Wildcards Under UNIX

by Anthony Skjellum

The Unix shell provides wildcard filename expansion for programs at execution. However, no standard facility was provided to expand wildcard filenames from within a program. This package provides the capability to expand wildcard expressions from within a Unix program. Thus, programs which accept/require wildcard filename expressions may be made interactive, if desired.

Description

The **Wild** package consists of five subroutines. These subroutines provide the ability to specify a wildcard expression and then request the name of each unambiguous file satisfying that expression, one at a time. This provides a low-level mechanism which may be elaborated upon in specific programs.

The essential mechanism for wildcard expansion involves the use of the standard Unix LS(1) (directory lister) facility in a subprocess. LS is used to create a list of file names, which are then read from a temporary file as needed.

Tutorial Introduction

Wild allows a program to handle one or more wildcard expressions concurrently. Each active wildcard expansion uses a file, so the number is limited to the number of files which a given process may have open at a given time. Normally, programs will only perform one wildcard expansion at a time.

C programs which use Wild should include the following:

```
#include <stdio.h>
#include <wild.h>
```

which includes the standard library header *(stdio.h)* and Wild's header *(wild.h)*. Successful compilation will not occur unless these statements are present near the beginning of each C program which uses Wild.

Each unique wildcard process requires a structure of type *WILD*. Most of the internal parts of this structure won't concern users. For each concurrent wildcard expansion to be performed, include a declaration of the form *WILD wild;* in the appropriate routine of the C program. This declares the structure *wild* in a form suitable for use by this package.

It will be up to the user program to obtain or generate wildcard expressions. These will most often result from a user request (e.g., the program may obtain such an expression when it requests a file list). In any case, the subroutine *wopen* will be used to initiate wildcard expansion once the wildcard expression is available. *wopen*'s sole argument is a pointer to the string containing the wildcard expression. It, in turn, returns the information that will be needed for actual wildcard expansion. Calls to *wopen* take the form:

```
wild = wopen(expr);
```

where *wild* is a structure declared previously and *expr* is an appropriate wildcard expression. Conceptually, we provide the expression through *expr* and certain parameters are placed in *wild* as a result. The *wild* structure will be used by the other subroutines in this package to do the actual work of wildcard expansion.

A complementary function, called *wclose*, is provided to terminate wildcard processing. This is used when wildcard processing is completed and we wish to release the file pointer associated with the wildcard expansion. Always issue a *wclose* call for each open wildcard expansion before the program exits.

wclose takes a single argument which is a pointer to a structure of type *WILD*. Thus to close the expansion process *wild* we've used above, the call *wclose (&wild);* would be executed.

We have now seen how to open wildcard processing and terminate it. Now we'll discuss the most important subroutine in this package: *wnext* which actually supplies the filenames which have matched the specification given to *wopen*. Again, we will specify a pointer to the structure of type *WILD* (e.g., *error = wnext(&wild);*). *wnext* returns a completion value which is zero for successful completion. A non-zero value means that no filenames remain.

Once the call to *wnext* has been completed, the filename is available (if the call was successful). This is accessed as a component of the *WILD* structure. Thus, *wild.fnnam*[ ] is the string containing the filename just obtained. This may be manipulated as desired but will be overwritten during the next successful call to *wnext*.

The three subroutines already discussed allow for wildcard processing in C programs. One further subroutine, *wwind* allows the process to be restarted without the need to use *wclose* followed by a new call to *wopen*. (Should a program need to scan through the list of file names several times, it may be more advantageous to store them in a string array.) Note that, like *wnext*, *wwind* takes a pointer to type *WILD* as its argument.

This concludes the tutorial introduction. It is recommended that the reader study the file *wtest.c* which is an example of the Wild package in use.

More on Subroutines

wopen: initialization
The subroutine *wopen* must be called to open the wildcard processing for a given specification.

```
wopen(dir)
char *dir;
```

wpopen: alt. initialization
Same as *wopen* but uses a pipe to effect processing. This may prove faster than *wopen* but wildcard processing with *wpopen* may not be restarted with *wwind*.

```
wpopen(dir)
char *dir;
```

wclose: termination subroutine
The subroutine *wclose* is called when the processing of a given wildcard is completed.

```
wclose(wild)
WILD *wild;
```

wnext: get next matching filename
The subroutine *wnext* returns the next filename as *wnext.fnnam*[ ] if one is available. It returns zero for successful completion.

```
wnext(wild)
WILD *wild;
```

wwind: restart wildcard processing
This allows a rescan as described in the tutorial. This may only be used with processing initiated with *wopen*.

```
wwind(wild)
WILD *wild;
```

Linking

To link in the Wild subroutines, use the command *-lwild* as part of the link command line.

Copyright © 1982, A. Skjellum. All rights reserved. May be used for any non-commercial purposes. May be freely distributed for any non-commercial purpose but all copyright notices must be retained. Anthony Skjellum, 1695 Shennandoah Road, San Marino, CA 91108.

Wildcard Expressions

Wildcard expressions consist of any collection of regular expressions separated by spaces. Any file specifications that work with LS(1) work with this package. The following is an example of a valid wildcard expression:

```
*.c *.o *.s
```

which specifies all files ending in .c, .o, and .s in the current dictionary.

Limitations

File names were arbitrarily limited to 128 characters in length.

Files

/u1/src/Wild/wtest.c
/u1/lib/libwild.a

(Listing begins on page 14)

WILD Listing

Synopsis

```
#include <wild.h>
```

Subroutines

wopen: Initializer

```
WILD wopen(dir)
char *dir;
```

wpopen: Alt. Initializer

```
WILD wpopen(dir)
char *dir;
```

wclose: signals completion

```
wclose(wild)
WILD *wild;
```

wnext: get next file name

```
wnext(wild)
WILD *wild;
```

wwind: permits rescan of file names

```
wwind(wild)
WILD *wild;
```

type definitions

```
typedef struct      {
    char            W_actv;         /* signals initialization */
    char            W_fnam[128];    /* contains current file name */
    char            W_nam[128];     /* contains name of temp file */
    char            W_eol;          /* indicates out of names */
    FILE            *W_fp;          /* pointer to temp file */
    } WILD;
```

WILD Listing

(Text begins on page 12)

```
/*****************************************************************
*                                                                *
*       _wild.h -- header file for wildcard processing           *
*                                                                *
*       component of WILD --- package to provide wildcard        *
*       processing within a Unix(VII) program.                   *
*                                                                *
*       by Anthony Skjellum.                                     *
*       Copyright 1982 (c) All rights reserved.                  *
*                                                                *
*       May be used for all non-commercial purposes              *
*       and freely distributed for non-commercial                *
*       purposes as long as all copyright notices                *
*       are retained.                                            *
*                                                                *
*****************************************************************/


typedef struct {

        char    W_actv; /* active indicator */
        char    W_fnam[128]; /* string for current file name */
        char    W_nam[128];  /* string containing file name which
                                has the list of entries.  Used for
                                deletion purposes. */
        char    W_eol;  /* indicates that list exhausted */
        FILE    *W_fp;  /* file ptr to files containing file names */

        } WILD;




/*****************************************************************
*                                                                *
*       wild.h -- header file for wildcard processing            *
*                                                                *
*       component of WILD --- package to provide wildcard        *
*       processing within a Unix(VII) program.                   *
*                                                                *
*       by Anthony Skjellum.                                     *
*       Copyright 1982 (c) All rights reserved.                  *
*                                                                *
*       May be used for all non-commercial purposes              *
*       and freely distributed for non-commercial                *
*       purposes as long as all copyright notices                *
*       are retained.                                            *
*                                                                *
*****************************************************************/

#include <_wild.h>



        WILD wopen(); /* define this as returning struct of type
                         WILD */

        WILD wpopen(); /* define this as returning struct of type
                          WILD */
```

(Continued on column 2)

```
#include <stdio.h>
#include <_wild.h>

/*

        program to provide wildcard processing from
        within a Unix program.

        by Anthony Skjellum (c) 1982.
        May be used for any non-commerical purpose.  Commercial
        uses without author's written permission are prohibited.

        Wild functions:

        wopen(dir) ---  initiate wildcard processing on reg. expression
                        dir.

                        This subroutine returns a structure of type
                        WILD.  The structure of type WILD contains
                        information to be used when actually requesting
                        files from the expanded specification one at
                        a time (using wnext()).

        wpopen(dir) --- initiate wildcard processing on reg. expression
                        dir.  This alternate initializer uses a pipe
                        to obtain wild card expansion data.  It may
                        prove to be faster, but the wwind() fn is not
                        usable when wpopen is used to open a wildcard
                        expansion.

                        Same functional particulars as wopen().


        wnext(wild) --  get the next file name from the preinitialized
                        wild card sequence indicated by wild (ptr to
                        type WILD).

                        wild.fname contains the current name.
                        wild.eol   is -1 if the list is exhausted.

        wwind(wild) --  reset the list to the beginning for additional
                        processing.  This allows multiple scans through
                        the same wild card spec. without reusing wild().
                        This won't work if wpopen() was used to initiate the
                        processing.

        wclose(wild) -  closes the wildcard spec.  Makes it no longer
                        available for use.  This call should be used
                        on each open wildcard spec. before the program
                        terminates.  Otherwise, unnecessary temp. files
                        will accumulate on /tmp between purges.


*/

/* wopen(dir) */

WILD wopen(dir)
char *dir; /* regular expression */
{
        int i;
        FILE *fp; /* temporary */
        WILD wtemp; /* temp. struct. returned to caller */

        char str[30];
        char temp[100]; /* command string */
```

(Continued on column 1, page 16)

WILD Listing

(Listing continued, text begins on page 12)

```
        strcpy(str,"/tmp/dirXXXXXX"); /* will be turned into
                                         unique file name */

        mktemp(str); /* make into unique file name */

        sprintf(temp,"sh -c \"/bin/ls %s)%s 2> /dev/null\"",dir,str);
        system(temp); /* exec call */

        fp = fopen(str,"r"); /* open for reading */

        if(fp == NULL) /* error */
        {
           fprintf(stderr,"wopen: cannot access created temp. file %s\n",str);
           exit(1);
        }

        wtemp.W_fp = fp; /* save */
        wtemp.W_actv = 1; /* make active */
        wtemp.W_fnam[0] = '\0';
                        /* make null ie no file name until on reqested */
        wtemp.W_eol = 0; /* not exhausted state */

        strcpy(wtemp.W_nam,str); /* save file name for deletion when
                                    done */

        /* we're done */

        return(wtemp); /* return the structure */



}

/* wclose(&wtemp) */

wclose(wtemp)
WILD *wtemp;
{




        wtemp->W_actv = 0; /* make inactive */

        unlink(wtemp->W_nam); /* remove the file */
        fclose(wtemp->W_fp); /* and close it for good measure */

}



/* wwind(&wtemp) */

wwind(wtemp)
WILD *wtemp;
{
        lseek(wtemp->W_fp->_file,0L,0); /* rewind file */
        wtemp->W_eol = 0; /* reset eol condition */

        return(0);
}

/* wnext(&wtemp) */
```

```
wnext(wtemp)
WILD *wtemp;
{
        char string[200];
        int error;

        error = fscanf(wtemp->W_fp,"%s",string); /* get a line, essentially */



        if(error<=0) /* eof or other problem: assume eof */
        {
                wtemp->W_eol = 1; /* set eol cond */
                return(-1); /* indicate */

        }

        if(strlen(string)>127) /* error */
        {
                fprintf(stderr,"wnext: error file name too long\n");
                exit(1);
        }

        strcpy(wtemp->W_fnam,string); /* copy */
        return(0);

}

/* wpopen(dir) */

WILD wpopen(dir)
char *dir; /* regular expression */
{
        FILE *popen();

        int i;
        FILE *fp; /* temporary */
        WILD wtemp; /* temp. struct. returned to caller */

        char str[30];
        char temp[100]; /* command string */

        sprintf(temp,"sh -c \"/bin/ls %s 2> /dev/null\"",dir);
        fp = popen(temp,"r"); /* open reading pipe */

        if(fp == NULL) /* error */
        {
                fprintf(stderr,"wpopn: cannot make pipe\n");
                exit(1);
        }



        wtemp.W_fp = fp; /* save */
        wtemp.W_actv = 2; /* make active: and indicate pipe used */
        wtemp.W_fnam[0] = '\0';
                        /* make null ie no file name until on reqested */
        wtemp.W_eol = 0; /* not exhausted state */

        wtemp.W_nam[0] = '\0';  /* make null for pipe */

        /* we're done */

        return(wtemp); /* return the structure */

}
```

(Continued on page 43, column 1)

Tests for Pidgin

This note introduces seven test programs designed to test an implementation of Pidgin. Passing all of these tests probably means that only a few macro bugs are left, or none if you are lucky. They form a useful functional definition of the language, even if not quite complete.

I would be delighted to hear from anyone with a macro bug that was not caught by these tests, so that a test for it could be added. (Such a bug would show, if the previously published code for Tincmp, for instance, failed to run even though all these tests were passed.)

The first program looks very simple, yet when it runs, your job is at least half done. It basically shows that you have a compiler capable of accepting an input and producing a running output.

The second program introduces the test methodology used in the remaining programs. The IF and ENDIF macros must work for the tests to be useful. Bugs on IF . . . ENDIF usually result in completely wild response and so are easy to see. The messages printed out if everything works right will just give sequences of numbers, then of letters. These turned out to be helpful in identifying pieces of code in the machine. Sequences of nine identical characters stand out fairly well, and you can locate parts of code for single step checking. If something wrong happens, a brief message is put out between the blocks of numbers.

The second program tests byte arithmetic, and the third tests integer addition and subtraction. Special tests are made for cases where there might be some try at optimization. Test 4 exercises the multiplication and division, with several cases covering the good possibility that these are done by subroutine. The logical operations are also checked.

Test 5 is a test of subscripting and of the WHILE control statement. Test 6 tests the remaining control structures and pack and unpack.

Test 7 is the most difficult of the individual tests. It is very specific about what needs to happen for the I/O statements, and requires taking out idiot insurance against such things as writing on files opened to read, and vice versa. Since these functions are almost surely done by subroutine, they probably require the most assembly language programming.

These seven test programs for Pidgin are public domain. Machine readable copies for Apple are available for the $5 cost of a disk and mailing from Bootstrap Toolkit, 439 S. Orange Ave., S. Orange, NJ 07079. ▶▶J

(Listing begins at right)

by William A. Gale

William A. Gale, Ph.D., 439 South Orange Avenue, South Orange, NJ 07079.

Pidgin Test #1

(Text begins on page 18)

Listing One

```
top
lomem=04096
himem=36000
register=086
beginmain(ac,iav)
ms 'hello wor'
ms 'ld. ms is'
ms ' working.'
endmain
bottom
```

Pidgin Test #2

Listing Two

```
top
lomem=04096
himem=36000
register=086
byte xx
byte aa
byte bb
byte cc
byte dd
byte ee
beginmain(ac,iav)
        ms 'begin t02'
aa=+001
bb=+002
xx=aa==bb
if xx
        ms '==fails!='
        ms 'or if  or'
        ms 'definitin'
else
        ms '111111111'
endif
bb=+001
```

(Continued on column 2)

```
xx=aa==bb
if xx
        ms '222222222'
else
        ms '==fails=='
endif
xx=aa!=bb
if xx
        ms '!=fails=='
endif
ms '333333333'
bb=+005
xx=aa!=bb
if xx
else
        ms '!=fails!='
endif
ms '444444444'
bb=+001
xx=aa<!bb
if xx
        ms '<!fails=='
endif
ms '555555555'
bb=+002
xx=bb<!aa
if xx
        ms '<!fails>!'
endif
ms '666666666'
xx=aa<!bb
if xx
else

        ms '<!fails<!'
endif
ms '777777777'
xx=bb<=aa
if xx
        ms '<=fails>!'
endif
ms '888888888'
bb=+001
```

(Continued on column 3)

```
xx=aa<=bb
if xx
else
        ms '<=fails=='
endif
ms '999999999'
bb=+002
xx=aa<=bb
if xx
else
        ms '<=fails<!'
endif
ms '000000000'
aa++
xx=aa!=bb
if xx
        ms '++fails..'
endif
ms 'aaaaaaaaa'
aa++
aa--
xx=aa!=bb
if xx
        ms '--fails..'
endif
ms 'bbbbbbbbb'
aa=+050
bb=+064
dd=aa+bb
ee=+114;2
xx=dd!=ee
if xx
        ms 'add...+++'
endif
ms 'ccccccccc'
cc=+000
bb=+050
dd=bb-aa
aa=+000;0
xx=aa!=dd
if xx
        ms 'sub...+-+'
```

(Continued on next page)

Pidgin Test #2

(Listing continued, text begins on page 18)

Listing Two

```
endif
ms 'ddddddddd'
aa=+000;0
bb=+050
cc=aa-bb
bb=+064
dd=aa-bb
bb=+114;2
ee=aa-bb
bb=cc+dd
xx=bb!=ee
if xx

        ms 'add...-+-'
endif
ms 'eeeeeeeee'
bb=cc-dd
ee=+014;0
xx=ee!=bb
if xx
        ms 'sub---=>+'
endif
ms 'fffffffff'
bb=dd-cc
aa=+000;0
cc=aa-ee
xx=bb!=cc
if xx
        ms 'sub---=>-'

endif
ms 'ggggggggg'
aa=+001
bb=+003
aa=aa+bb
cc=+004;2
xx=cc!=aa
if xx
        ms 'add.x=x+.'
```

Pidgin Test #3

Listing Three

```
top
lomem=04096
himem=36000
register=086
byte xx
int iaa
int ibb
int icc
int idd
int iee
beginmain(ac,iav)
        ms 'begin t03'
iaa=+10001
ibb=+10002
xx=iaa==ibb
if xx

        ms '==fails!='
        ms 'or if  or'
        ms 'definitin'
endif
ms '111111111'
ibb=+10001
xx=iaa==ibb
if xx
else
        ms '==fails=='
endif
ms '222222222'
xx=iaa!=ibb
if xx
        ms '!=fails=='
endif
ms '333333333'
ibb=+10005
xx=iaa!=ibb
if xx
else
        ms '!=fails!='
endif
ms '444444444'
ms 'aaaaaaaaa'
iaa++
iaa--
xx=iaa!=ibb
if xx
        ms '--fails..'
endif
ms 'bbbbbbbbb'
iaa=+10050
ibb=+10064
idd=iaa+ibb
iee=+20114;2
xx=idd!=iee
if xx
        ms 'ads...+++'
endif
ms 'ccccccccc'
icc=+10000
ibb=+10050
idd=ibb-iaa
iaa=+00000
xx=iaa!=idd
if xx
        ms 'sub...+-+'
endif
ms 'ddddddddd'
iaa=+00000
ibb=+10050
icc=iaa-ibb
ibb=+10064
idd=iaa-ibb
ibb=+20114;2
iee=iaa-ibb
ibb=icc+idd
xx=ibb!=iee
if xx
        ms 'ads...-+-'
endif
ms 'eeeeeeeee'
ibb=icc-idd
iee=+00014;0
xx=iee!=ibb
if xx
        ms 'sub---=>+'
```

```
endif
ms 'hhhhhhhhh'
bb=aa+bb
cc=+007;3
xx=cc!=bb
if xx
        ms 'add.x=.+x'
endif
ms 'iiiiiiiii'
cc=cc+cc
bb=+014;6
xx=bb!=cc
if xx
        ms 'add.x=x+x'
endif
ms 'jjjjjjjjj'
aa=+009
bb=+005
aa=aa-bb
cc=+004;0
xx=cc!=aa
if xx
        ms 'sub.x=x-.'
endif
ms 'kkkkkkkkk'
aa=+009
bb=aa-bb
xx=cc!=bb
if xx
        ms 'sub.x=.-x'
endif
ms 'lllllllll'
aa=aa-aa
bb=+000;0
xx=aa!=bb
if xx
        ms 'sub.x=x-x'

endif
        ms 'endof t02'
endmain
bottom
```

(Continued on page 20, column 2)

```
ibb=+10001
xx=iaa<!ibb
if xx
        ms '<!fails=='
endif
ms '555555555'
ibb=+10002
xx=ibb<!iaa
if xx
        ms '<!fails>!'
endif
ms '666666666'
xx=iaa<!ibb
if xx
else
        ms '<!fails<!'
endif
ms '777777777'
xx=ibb<=iaa
if xx
        ms '<=fails>!'
endif
ms '888888888'
ibb=+10001
xx=iaa<=ibb
if xx
else
        ms '<=fails=='
endif
ms '999999999'
ibb=+10002
xx=iaa<=ibb
if xx
else
        ms '<=fails<!'
endif
ms '000000000'
iaa++
xx=iaa!=ibb
if xx
        ms '++fails..'

endif
```

(Continued on page 20, column 3)

```
endif
ms 'fffffffff'
ibb=idd-icc
iaa=+00000
icc=iaa-iee
xx=ibb!=icc
if xx
        ms 'sub---=>-'
endif
ms 'ggggggggg'
iaa=+10001
ibb=+10003
iaa=iaa+ibb
icc=+20004;2
xx=icc!=iaa
if xx
        ms 'ads.x=x+.'
endif
ms 'hhhhhhhhh'
ibb=iaa+ibb
icc=+30007;3
xx=icc!=ibb
if xx
        ms 'ads.x=.+x'
endif
ms 'iiiiiiiii'
icc=icc+icc
ibb=+60014;6
xx=ibb!=icc
if xx
        ms 'ads.x=x+x'
endif
ms 'jjjjjjjjj'
iaa=+10009
ibb=+10005
iaa=iaa-ibb
icc=+00004;0
xx=icc!=iaa
if xx

        ms 'sub.x=x-.'
endif
ms 'kkkkkkkkk'
```

(Continued on next page)

Pidgin Test #3

(Listing continued, text begins on page 18)

Listing Three

```
iaa=+10009
ibb=iaa-ibb
xx=icc!=ibb
if xx
        ms 'sub.x=.-x'
endif
ms '111111111'
iaa=iaa-iaa
ibb=+00000
xx=iaa!=ibb
if xx
        ms 'sub.x=x-x'
endif
        ms 'endof t03'
endmain
bottom
```

Pidgin Test #4

Listing Four

```
top
lomem=04096
himem=36000
register=086
byte aa
int iaa
byte bb
int ibb
byte cc
int icc
int idd
byte dd
beginmain(ac,iav)
        ms 'begin t04'
aa=+003
bb=aa
cc=aa!=bb
if cc
iaa=+00005
icc=-iaa
cc=icc!=ibb
if cc
        ms 'int=-int.'
endif
ms '777777777'
iaa=+10256
ibb=+00003
icc=iaa*ibb
idd=+30768
cc=idd!=icc
if cc
        ms 'intmul+*+'
endif
ms '888888888'
ibb=-ibb
icc=iaa*ibb
idd=-idd
cc=idd!=icc
if cc
        ms 'intmul+*-'
endif
ms '999999999'
iaa=+00030
ibb=+00040
iaa=iaa*ibb
icc=+01200
aa=icc!=iaa
if aa
        ms 'mul.x=x*.'
endif
ms '000000000'
iaa=+00030
ibb=iaa*ibb
aa=icc!=ibb
if aa
        ms 'mul.x=.*x'
endif
ms 'aaaaaaaaa'
iaa=iaa*iaa
icc=+00900
aa=iaa!=icc
if aa
ms 'ggggggggg'
ibb=+00017
icc=iaa/ibb
idd=+00603
cc=icc!=idd
if cc
        ms 'frcdiv+/+'
endif
ms 'hhhhhhhhh'
iaa=-iaa
icc=iaa/ibb
idd=-idd
cc=icc!=idd
if cc
        ms 'frcdiv-/+'
endif
ms 'iiiiiiiii'
aa=+001
bb=+001
cc=aa&bb
if cc
else
        ms '"&" or if'
endif
ms 'jjjjjjjjj'
bb=+000
cc=aa&bb
if cc
        ms '"&"(orif)'
endif
ms 'kkkkkkkkk'
cc=aa?bb
if cc
else
        ms '"?"...101'
endif
ms '111111111'
aa=+000
cc=aa?bb
if cc

        ms '"?"...000'
endif
ms 'mmmmmmmmm'
cc=!aa
```

```
        ms 'byte tran'
endif
ms '111111111'
iaa=+12345
ibb=iaa
cc=iaa!=ibb
if cc
        ms 'int trans'
endif
ms '222222222'
icc=+00235
bb=icc
aa=+235
cc=aa!=bb
if cc
        ms 'int to by'
endif
ms '333333333'
ibb=aa
icc=+00100
aa=+100
ibb=aa
cc=ibb!=icc
if cc
        ms 'byt to in'
endif
ms '444444444'
iaa=+00000
ibb=+00005
ibb=iaa-ibb
cc=ibb<=iaa
if cc
else
        ms 'int<=.-<0'
endif
ms '555555555'
cc=ibb<!iaa
if cc
else
        ms 'int<!.-<0'
endif
ms '666666666'
```

(Continued on page 22, column 2)

```
        ms 'mul.x=x*x'
endif
ms 'bbbbbbbbb'
iaa=+30003
ibb=+00003
iaa=iaa/ibb
icc=+10001
aa=iaa!=icc
if aa
        ms 'div.x=x/.'
endif
ms 'ccccccccc'
ibb=iaa/ibb
icc=+03703
aa=icc!=ibb
if aa
        ms 'div.x=./x'
endif
ms 'ddddddddd'
icc=icc/icc
iaa=+00001
aa=iaa!=icc
if aa
        ms 'div.x=x/x'
endif
ms 'eeeeeeeee'
iaa=+10256
ibb=+00016
icc=iaa/ibb
idd=+00641
cc=idd!=icc
if cc
        ms 'intdiv+/+'
endif
ms 'fffffffff'
ibb=-ibb
icc=iaa/ibb
idd=-idd
cc=idd!=icc
if cc
        ms 'intdiv+/-'
endif
```

(Continued on page 22, column 3)

```
if cc
else
        ms '"!"....01'
endif
ms 'nnnnnnnnn'
bb=+001
cc=aa-bb
dd=!cc
if dd
        ms '"!"....10'
endif
        ms 'endof t04'
endmain
bottom
```

Pidgin Test #5

Listing Five

```
top
lomem=04096
himem=36000
register=086
byte aa(100)
byte bb(00300)
int iaa(100)
int ibb(00300)
byte cc
byte dd
int icc
byte ss
byte ee
byte ff
int idd
int iee
int iff
beginmain(ac,iav)
        ms 'begin t05'
goto 10
        ms 'gotofails'
loc 20
goto 30
```

(Continued on next page)

Pidgin Test #5

(Listing continued, text begins on page 18)

Listing Five

```
        ms 'and again'
loc 10
goto 20
loc 30
ms '111111111'
cc=+001
dd=+000
aa(dd)=cc
ee=aa(dd)
ff=ee!=cc
if ff
        ms 'byt...byt'
endif
ms '222222222'
dd=+099
aa(dd)=cc
ee=aa(dd)
ff=ee!=cc
if ff
        ms 'byt.99byt'
endif
ms '333333333'
ss=+000
while
        dd--
        ee=+000
        ff=dd!=ee
on ff
        aa(dd)=ss
        ee=aa(dd)
        ff=ee!=ss
        if ff
                ms 'byt.i.byt'
                ms 'or while.'
                goto 51
        endif
        ss++
endwhile
ms '444444444'
                ms 'int(int).'
                goto 53
        endif
endwhile
ms '666666666'
loc 53
dd=+100
cc=+000
iee=+10000
while
        ff=cc!=dd
on ff
        iaa(cc)=iee
        iee++
        cc++
endwhile
cc=+000
iee=+10000
while
        ff=cc!=dd
on ff
        idd=iaa(cc)
        ff=idd!=iee
        if ff
                ms 'int(byte)'
                goto 54
        endif
        iee++
        cc++
endwhile
ms '777777777'
loc 54
ms 'endof t05'
endmain
bottom
```

Pidgin Test #6

Listing Six

```
top
lomem=04096
himem=36000
register=086
ms '555555555'
d4=+004
choose on d4
case a1
        ms '4!=1.....'
case b2
        ms '4!=2.....'
case c3
        ms '4!=3.....'
default
endchoose
ms '666666666'
d4=+003
choose on d4
case a1
        ms '3!=1.....'
case b2
        ms '3!=2.....'
case c3
default
        ms 'default..'
endchoose
ms '777777777'
goto 01
loc 01
goto 02
loc 02
ms '888888888'
a1=+000
b2=+005
pack(iax,a1,b2)
iaz=+00005
c3=iax!=iaz
if c3
        ms 'packfails'
endif
ms '999999999'
a1=+100
pack(iax,a1,b2)
unpack(iax,c3,d4)
e5=a1!=c3
c3=b2!=d4
e5=e5?c3
if e5
```

```
loc 51
ee=+098
ff=ee!=ss
if ff
        ms 'while....'
endif
idd=+00300
icc=+00000
while
        ff=icc!=idd
on ff
        dd++
        bb(icc)=dd
        ee=bb(icc)
        ff=ee!=dd
        if ff
                ms 'byt.i.int'
                goto 52
        endif
        icc++
endwhile
ms '555555555'
loc 52
icc=+00000
iee=+00100
while
        ff=icc!=idd
on ff
        iee++
        ibb(icc)=iee
        icc++
endwhile
iee=+00100
icc=+00000
while
        ff=icc!=idd

on ff
        iee++
        iff=ibb(icc)
        icc++
        ff=iff!=iee
        if ff
```

(Continued on page 24, column 2)

```
byte a1
byte b2
byte c3
byte d4
byte e5
int iax
int iay
int iaz
beginmain(ac,iav)
        ms 'begin t06'
a1='0'
a1++
a1++
a1++
b2='3'
c3=a1!=b2
if c3
        ms 'charcons.'
endif
ms '111111111'
d4=+001
gosub 01
ms '222222222'
if d4
        ms 'globalvbl'
endif
ms '333333333'
gosub 02
ms '444444444'
a1=+001
b2=+002
c3=+003
d4=+001
choose on d4
case a1
case b2

        ms '1!=2.....'
case c3
        ms '1!=3.....'
default
        ms 'default..'
endchoose
```

(Continued on page 24, column 3)

```
        ms 'unpack...'
endif
        ms 'endof t06'
stop 1
ms '!!!!!!!!!'
endmain
sub 02
ms 's2s2s2s2s'
a1=+001
if a1
        return
endif
ms 'return...'
endsub
sub 01
ms 'slslslsls'
d4=+000
endsub
bottom
```

Pidgin Test #7

Listing Seven

```
top
himem=36863
lomem=04096
register=086

byte c0
byte c1
byte c2
byte c3
byte aa
byte bb
byte cc
byte dd
byte ee
byte tr
int ibc
int iaa
int ibb
int icc
```

(Continued on next page)

Pidgin Test #7

(Listing continued, text begins on page 18)

Listing Seven

```
byte f1(00300)
byte f2(00300)
byte nl
beginmain(ac,iav)
        ms 'begin t07'

c0=+000
c1=+001
c2=+002
c3=+003
close f2
aa=c2!=er
if aa
        ms 'closclosd'
        ;'closing a file not
        ;opened to return error'
endif
ms '111111111'
tr='r'
ibc=+00150
open f1 for tr at ibc
aa=c0!=er
if aa
        ms 'cant open'
        ;'because open returns
        ;an error'
        stop 1
endif
ms '222222222'
open f1 for tr at ibc
aa=c2!=er
tr='w'
ibc=+00002
open f1 for tr at ibc
bb=c2!=er
aa=aa?bb
if aa
        ms 'openopend'
        ;'opening an opened
open f2 for tr at ibc
aa=c0!=er
if aa
        ms 'cantopenw'
        ;'cant open f2
        ;for writing'
        stop 2
endif
ms '888888888'
er=c0
read cc from f2
aa=er!=c2
if aa
        ms 'rd on wr.'
        ;'read on write
        ;opened should set er=2'
endif
ms '999999999'
cc=c1
er=c0
write cc into f2
aa=er!=c0
if aa
        ms 'wr set er'
        ;'so not sure that
        ;we can write'
endif
ms '000000000'
aa=cc!=c1
if aa
        ms 'wr alt vb'
        ;'write altered the
        ;variable being written'
endif
ms 'aaaaaaaaa'
er=c2
cc=c2
write cc into f2
aa=er!=c0
if aa
        ms 'wr xseter'
        ;'write didnt set
        ;er to zero'
        stop 3
endif
ms 'eeeeeeeee'
ibc=+00250
tr='r'
open f2 for tr at ibc
aa=er!=c0
if aa
        ms 'cantreopn'
        ;'got an error trying
        ;to open a closed file'
        stop 4
endif
ms 'fffffffff'
iaa=c1
ibb=+00514
cc=c1
while
        read bb from f2
        aa=bb!=cc
        if aa
                ms 'cantrered'
    ;'the input differs from
    ;the output'
                goto 40
        endif
        aa=iaa<!ibb
on aa
        cc++
        iaa++

endwhile
ms 'gggggggg'
read aa from f2
cc=c0-c1
bb=aa!=cc
cc=er!=c1
bb=bb?cc
if bb
    ms 'eof fails'
    ;'must have both
    ;char = -1 and er=1 on eof'
```

```
        ;file to return error'
         stop 2
endif
ms '333333333'
aa='1'
cc=+000
er=c1
read cc from f1
bb=cc!=aa
if bb
        ms 'readfails'
        ;'to return a character'
endif
ms '444444444'
aa=c0!=er
if aa
        ms 'rd set er'
        ;'read doesn"t set
        ;er to zero'
endif
ms '555555555'
write cc into f1
aa=c2!=er
if aa
        ms 'wr on rd.'
endif
ms '666666666'
er=c0
read cc from f1
aa='2'
bb=aa==cc
aa=er!=c0
aa=aa&bb
if aa
        ms 'rdunseter'
        ;'read set er to
        ;non zero but read
        ;the correct char'
endif
ms '777777777'
tr='w'
ibc=+00250
```

(Continued on page 24, column 2)

```
endif
ms 'bbbbbbbbb'
aa=c0
while
        read cc from f1
        bb=er==c1
        if bb
                goto 10
        endif
        aa++
        bb=aa!=c0
on bb
endwhile
ms 'noeofread'
;'we didnt find an
;end of file in 256
;characters of reading f1'
loc 10
ms 'ccccccccc'
iaa=c1
ibb=+00512
cc=c2
while
        cc++
        write cc into f2
        aa=er!=c0
        if aa
                ms 'writerror'
                goto 20
        endif
        aa=iaa<!ibb
on aa
        iaa++
endwhile
loc 20
ms 'ddddddddd'
close f2
aa=er!=c0
if aa
        ms 'cantclose'
        ;'we get an error when
        ;trying to close f2'
```

(Continued on pge 24, column 3)

```
endif
ms 'hhhhhhhhh'
loc 40
ms 'please ty'
ms 'pe a line'
nl=+141
write nl
icc=+00000
while
        read bb
        write bb
        aa=bb!=nl


on aa
        f1(icc)=bb
        icc++

endwhile
ibb=+00000
while
        aa=ibb<!icc
on aa
        aa=f1(ibb)
        write aa
        ibb++
endwhile
write nl
ms 'endof t07'
endmain
bottom
```

End Listing

A Novice Builds the Sinclair ZX81

Sinclair offers its ZX81 computer in kit form for those into electronic kit building or saving money. I fall into the latter category – the idea of saving (at that time) $50 on a microcomputer appealed to me. The drawback was that I had no experience in computers or electronics. I had never soldered anything in my life, and I couldn't tell a resistor from a capacitor.

So what? How does all this relate to the reader? Why would a computer professional or a dedicated amateur be interested in the struggles of an electronic novice? Suppose your teenager wants your advice on the acquisition of an affordable computer. Your next door neighbor has been hinting that he might like to "fool around" with a computer. You are a wizard programmer and operator, but would like a chance to see what the insides of a microcomputer are like. If so, the ZX81 kit is of interest to you; take comfort in the fact that your son or neighbor will be able to do what a beginner like me has done.

Without further ado, I will describe my adventures building a computer from a kit.

Preliminaries

While waiting for UPS to deliver my kit, I ventured out to Radio Shack and bought a soldering iron. The salesman was very helpful and even maintained his composure when I told him I was building a computer and needed a soldering iron. After answering a few of his questions, he sold me a 30-watt iron, some .032 diameter rosin-core solder, desoldering braid, and a super-deluxe soldering iron holder. Thanks to his foresight and sales ability, I avoided at least three more trips to Radio Shack when I discovered that I really needed those things. Unfortunately, I still don't know what the diameter is .032 of . . . but it worked.

On to the kit! Although I am not an experienced electronic kit builder, I knew enough to thoroughly read the instructions before I did *anything* else. What a shock! Apparently, the instructions that come with electronics kits are different from the kind you get with model airplanes and children's bicycles. I expected step-by-step, fully-illustrated, foolproof instructions to lead me by the hand through the assembly process.

In reality, Sinclair's "ZX81 Assembly Instructions" contain only ten steps. Only four of the steps deal with the installation of components onto the circuit board. There are critical pieces of information hidden throughout the sections entitled *Precautions*, *Component Identification*, and *Circuit Board Assembly*. The instructions are obviously written by an Englishman and it took me a while to determine that a "Pozidrive screwdriver with a No. 1 point" meant a small Phillips head screwdriver. One line in the fault-finding section bears quoting. "If you find a bad joint or short, shame on you! You should have checked more closely the first time." With advice like that, how can you go wrong?

Once I got past the shock of not being spoonfed, I found the instructions to be complete and understandable. If clarification was required, Sinclair usually provided an illustration. The instructions included a drawing of the physical location of components on the circuit board and a circuit diagram. I recommend that Sinclair include a drawing of the printed wiring on the circuit board. This would allow verification of component placement and depict the relationship among parts. No doubt, the circuit diagram provides the same information, but it seems to be arranged according to function rather than physical location. At times, it is hard for the inexperienced kit builder to follow.

Having read the instructions, I pulled out the various parts and began the identification process. Not knowing exactly how to proceed, I pulled out a piece of paper and copied the parts list from the section entitled *Component List* and matched parts to the list. After the first time through the list, I found that I had more parts than called for in the instructions. Fortunately, I discovered the addendum page to the instructions which apply to the U.S. version of the kit. Armed with this additional information, I was able to account for the use of all but three parts. I was a little worried, but didn't let it slow me down. One minor point . . . resistor R-30 was missing from my kit and I had to make another quick trip to Radio Shack.

Construction

Confident that I had the proper equipment and all the necessary parts, I cautiously approached my first soldering operation. The instructions indicated that the order of installation should be, "resistors, capacitors and IC sockets." I was overcome by doubt. I didn't want to ruin any of the components or the circuit board by improper soldering, so I cheated a little bit. The IC sockets seemed to be fairly foolproof and I didn't see too much in them that could burn out. I soldered one of them in place and knocked off for the night. The next day I carried the board and newly soldered socket into work to be checked out by a friend who had done some soldering. He approved my work and reassured me. I started back to work on the kit.

The IC sockets presented no problem. One nice feature of the Sinclair kit revealed itself during this phase. The back of the circuit board was coated in plastic to prevent drips of solder from making contact between the printed circuits. A touch like this is an outstanding idea for use in a kit that might be assembled by amateurs.

Resistors proved to be a little frustrating. Sinclair took it upon itself to preclip the leads on about half of the resistors. Generally speaking, this was convenient; however, some of the resistors had to be installed vertically. When the clipped leads coincided with the vertical placement, I ran into trouble. On two occasions, I had to solder more wire onto a resistor lead so that it could be installed. This added work, which occurred as a result of an attempt by Sinclair to lighten my load, was only a minor set-back.

Capacitors were easy! Two of them had to be oriented according to their polarity, but the instructions were clear and the board properly marked. Even the diodes were simple to put in correctly. In general, the board markings which indicated positioning of the components were easy to read and understand. The exception occurred when I tried to place the additional parts required for the U.S. version. For some reason, the markings on the board left room for doubt as to which hole matched which markings. The solution proved to lie in the circuit diagram. I was able to determine that one end of resistor R-32 was tied into diode D-9 and pin 3 of the TV modulator, and I was in business. Again, a novice, I would have been more comfortable with a drawing of the circuit board as it is physically constituted in addition to the functional layout provided in the circuit diagram.

by John Connell, Jr.

John C. Connell, Jr., 144 Lexington Street, Middletown, RI 02840.

Although the resistors, capacitors, IC sockets, diodes and transistors represented only two sub-steps of the directions, they provided the bulk of the work and took the most time. There are 34 resistors, 14 capacitors, 5 integrated circuits with 140 pins all together, and 9 diodes. I don't want to forget to mention the two resistor packs. It was during the installation of the first of the resistor packs that I had to resort to the use of "assembly language." Fortunately, the children were in bed and did not hear their father in his role as a "cursor."

Prior to putting the ZX81 together, I had read two accounts of this operation by experienced kit builders. In each case, they warned me not to install the resistor packs backwards, as they had. The instructions warned me not to install the resistor packs "the wrong way round." I had been well warned! So, I very carefully soldered the first resistor pack into the circuit board backwards. Imagine how I felt. I have a weak excuse: there is an orientation mark on each resistor pack which is to be aligned with a corresponding mark on the circuit board. Proper installation will result in the mark on the resistor pack being placed away from the front of the board. I guess it is impossible for an American to get this right the first time. There is something unnatural in installing a component with markings where you can't read them! Oh well, Radio Shack had provided the desoldering braid and I managed to get my money's worth out of it.

With one exception, the remaining parts were easily installed. These parts consisted of power, ear and microphone jacks, TV modulator, power regulator, heat sink, keyboard connectors, keyboard and case. The U.S. version also included a channel select switch to allow for choice between channels 2 and 3. It was this switch that I installed incorrectly. It is designed to protrude below the bottom of the case; therefore, it must be soldered to the bottom of the circuit board. Naturally, I put the switch on the same side of the board as all the other components. Back to the desoldering braid. . . .

With all the parts soldered in place, it was a simple matter to push the ICs into their sockets and hook the keyboard up to the keyboard connectors on the circuit board. I was ready to "power up." (See how quickly I picked up the "technical" jargon!) First, though, I checked the placement and orientation of every component. I thoroughly checked the back of the board for solder bridges and shorts. Then I checked everything again with a magnifying glass. There was nothing left to do but try it.

I plugged in the TV connection and applied power . . . there were no loud noises and I couldn't see any smoke. I was encouraged. The next step was to tune the TV. I fiddled and fiddled, but couldn't get the "K" to appear in the lower left-hand corner of the TV screen. I was beginning to reach for the fault-finding section of the directions when I realized that the switch on the TV antenna box was in the "TV" position and not in the "Computer" position. Could this be the problem? Yes! That is all that was wrong. The computer worked! I had built a computer that worked! All that remained was to put the computer in its case and stick down the keyboard. This was child's play for the world's newest electronics whiz-kid.

Conclusions

The ZX81 kit took just over 6½ hours to build. About one hour of that was instruction reading and part identification. I believe that an experienced kit builder could put the ZX81 together in about 4 hours. In addition to acquiring a computer for less than $100, I learned quite a lot about electronics in general and the ZX81 computer in particular. I had fun! I had no idea that an activity like soldering small electronic parts onto a printed circuit board would be enjoyable. I would willingly put a ZX81 together for a friend just for the fun of it.

Since building the ZX81, I have had a chance to read the 164-page manual entitled *ZX81 BASIC Programming*. The book is written to be useful for both the novice and the individual familiar with BASIC or Z-80 Assembly Language. The more I experiment with programming, the greater my respect for the designers of the ZX81. It may be a little slow and not have all the features of the "mainframe" microcomputers, but it is a nifty little computer, and I expect a lot of time to pass before I reach the limits of its capabilities. Of course, the ZX81 only has 1K of RAM built in and I was able to use that up very quickly, but any day now UPS will deliver my 16K RAM Memory Module and I will be back in business.

DDJ

68000 Cross Assembler (XASM)

(Continued from October, Number 72)

Listing One

```
C
C AUTHOR:
C               Allen Kossow
C               Medical College of Wisconsin
C               Department of Physiology
C               8701 Watertown Plank Rd.
C               Milwaukee, WI  53226
C
C PGM NOTES:
C
C               Symbols are unique to 8 chrs.
C               Symbol table size is currently set to 512.
C               Symbol table size is determined by the memory constraints
C               of the machine it's running on.  The current symbol table
C               size works under RT-11 with 64kb of memory and under RSX-11.
C
C               This program follows most of the conventions of the
C               Motorola 68000 assembler with the following exceptions:
C
C               1) no macros or pc relative addressing (except branches)
C               2) generalized (i.e., simple) object file format
C               3) listing format similar to that of MACRO-11
C               4) doesn't handle adr regs as special cases (i.e., MOVEA is just a MOVE).
C
C
C               3) listing format similar to that of MACRO-11
C               4) doesn't handle adr regs as special cases (i.e., MOVEA
C                  is just a MOVE).
C
C


        OBJWC=OBJWC+1
        OBJBUF(1)=OPSK2+(OPNWRD(1).AND.'77)
        GOTO 7000
C
C +++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS MULT DIV AND CHK INSTRUCTIONS
C       <EA>,DN
C
C +++++++++++++++++++++++++++++++++++++++++++++++++
1100    IF(OP2EA.NE.1) GOTO 8500
        IF(OP1EA.EQ.2) GOTO 8500
        IF(OP1EA.EQ.0.OR.OP1EA.EQ.6) GOTO 1110
        IF(OP1EA.GT.6) GOTO 8500
        OPSKEL=OPSKEL+((OP1EA-1)*'10)+OP1DA
        GOTO 1120
1110    CALL PROCOP(OPNPTR)
        OPSKEL=OPSKEL+OPNWRD(1)
        OBJBUF(2)=OPNWRD(2)
        IF(OPNWC.EQ.2) OBJBUF(2) = OPNWRD(3)
        IF(OPNWC.EQ.2) OBJBUF(3) = OPNWRD(2)
        OBJWC = OBJWC + OPNWC
1120    OBJBUF(1)=OPSKEL+(OP2DA*'1000)
        GOTO 7000
C +++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS EXG INSTRUCTION
C       RX,RY
C
C +++++++++++++++++++++++++++++++++++++++++++++++++
1400    IF(OP1EA.EQ.0.OR.OP1EA.GT.2) GOTO 8500
        IF(OP2EA.EQ.0.OR.OP2EA.GT.2) GOTO 8500
        OPSKEL=OPSKEL+OP2DA
        OPSKEL=OPSKEL+(OP1DA *'1000)
        IF(OP1EA.EQ.1.AND.OP2EA.EQ.1) OBJBUF(1)=OPSKEL+'500
        IF(OP1EA.EQ.2.AND.OP2EA.EQ.2) OBJBUF(1)=OPSKEL+'510
        IF(OP1EA.EQ.OP2EA) GOTO 7000
        OBJBUF(1)=OPSKEL+'610
        GOTO 7000
C +++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS EXT AND SWAP INSTRUCTIONS
C       DN
C
C +++++++++++++++++++++++++++++++++++++++++++++++++
1500    IF(OPIDX.EQ.28) GOTO 1510
        IF(IMODE.EQ.3 ) OPSKEL = OPSKEL.OR.'100
1510    IF(OP1EA.NE.1) GOTO 8500
        OBJBUF(1)=OPSKEL+OP1DA
        GOTO 7000
C +++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS LEA INSTRUCTION
C       <EA>,AN
C
C +++++++++++++++++++++++++++++++++++++++++++++++++
1600    IF(OP1EA.EQ.0) GOTO 1610
        IF(OP1EA.EQ.3) GOTO 1620
        GOTO 8500
C
1610    CALL PROCOP(OPNPTR)
        OBJBUF(1) = OPSKEL.OR.OPNWRD(1).OR.OP2DA
        OBJBUF(2) = OPNWRD(2)
        IF (OPNWC.EQ.2) OBJBUF(2) = OPNWRD(3)
        IF (OPNWC.EQ.2) OBJBUF(3) = OPNWRD(2)
        OBJWC = OBJWC + OPNWC
        GOTO 7000
C
1620    OBJBUF(1) = OPSKEL.OR.OP2DA.OR.OP1DA
        OBJBUF(1) = OBJBUF(1).OR.((OP1EA-1)*'10)
        GOTO 7000
C
C +++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS LINK INSTRUCTION
C       AN, <DISPLACEMENT>
C
C +++++++++++++++++++++++++++++++++++++++++++++++++
1700    IF(OP1EA.NE.2.AND.OP2EA.NE.6) GOTO 8500
        CALL PROCOP(OPNPT2)
        IF (OPNWRD(3).EQ.0) GOTO 1710
        IF (OPNWRD(3).EQ.-1)GOTO 1710
        OP = OPNPTR
2110    CALL RLSTDC(OP,DLIST,ALIST)
        IF ((DLIST.EQ.0).AND.(ALIST.EQ.0)) GOTO 2150
C
C....   CHECK IF DESTINATION EA IS LEGAL FOR A STM INSTRUCTION
C....   -(AN) AND CTL ALTERABLE ADR MODES ARE LEGAL

C
        IF ((OP2EA.EQ.3).OR.(OP2EA.EQ.5)) GOTO 2112
        IF (OP2EA.EQ.0) GOTO 2112
        GOTO 8500
C
C....   REFORMAT DATA AND ADR BITMAPS FOR STM INSTRUCTION
C
2112    IF (OP2EA.NE.5) GOTO 2116
C
C....   -(AN) REQUIRES REGISTERS TO BE BACKWARDS IN THE BITMAP
C
        DLSTI = 0
        ALSTI = 0
        DO 2113,I=0,7
2113    IF((DLIST.AND.(2**I)).NE.0) DLSTI = (DLSTI.OR.(2**(7-I)))
C
        DO 2114,I=0,7
2114    IF((ALIST.AND.(2**I)).NE.0) ALSTI = (ALSTI.OR.(2**(7-I)))
C
        ALIST = DLSTI
        DLIST = ALSTI
C
C....   BUILD BITMAP
C
2116    CALL BLDMAP(DLIST,ALIST,OBJBUF(2))
C
C....   PROCESS DESTINATION OPERAND
C
        IF (OP2EA.EQ.0) GOTO 2118
C
C....   SIMPLE DESTINATION OPERAND
C
        OBJWC = 2
        OBJBUF(1) = OPSKEL.OR.OP2DA.OR.((OP2EA-1)*'10)
        GOTO 7000
C
C....   PROCESS COMPLEX DESTINATION OPERAND
C
2118    CALL PROCOP(OPNPT2)
        OBJBUF(1) = OPSKEL .OR. OPNWRD(1)
        OBJBUF(3) = OPNWRD(2)
        IF (OPNWC.EQ.2) OBJBUF(3) = OPNWRD(3)
        IF (OPNWC.EQ.2) OBJBUF(4) = OPNWRD(2)
        OBJWC = OBJWC + OPNWC
        GOTO 7000
C
C
C....   PROCESS LDM INSTRUCTION
C
C
2150    OPSKEL = OPSKEL.OR.'2000
```

```
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS INSTRUCTIONS OF FORM OPCODE <EA>
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C       CHK FOR CLR,NEG
1200    IF(OPIDX.EQ.18.OR.OPIDX.EQ.25) GOTO 1202
C       CHK FOR NOT,TST
        IF(OPIDX.EQ.27.OR.OPIDX.EQ.48) GOTO 1202
        IF(IMODE.NE.0) GOTO 8500                ! SIZE BITS ILLEGAL
        GOTO 1210
1202    IF(IMODE.EQ.1) GOTO 1205
        IF(IMODE.EQ.3) OPSKEL=OPSKEL+'200
        IF(IMODE.EQ.2.OR.IMODE.EQ.0)OPSKEL=OPSKEL+'100

1205    IF(OP1EA.EQ.0.OR.OP1EA.GE.6) GOTO 1210
        IF(OP1EA.GT.6) GOTO 8500
C
C....   PROCESS REG OPERAND
C
        OBJBUF(1)=OPSKEL+OP1DA+((OP1EA-1)*'10)
        GOTO 7000
C
C....   PROCESS COMPLEX OPERAND
C
1210    IF(OP1EA.NE.0.AND.OP1EA.NE.3) GOTO 8500
        IF(OP1EA.NE.3) GOTO 1215
        OBJBUF(1) = OPSKEL.OR.OP1DA.OR.'20
        GOTO 7000
C
C....   GENERATE EXTENSION WORDS AS NECESSARY
C
1215    CALL PROCOP(OPMPTR)
        OBJBUF(1) = OPSKEL.OR.OPMWRD(1)
        OBJBUF(2) = OPMWRD(2)
        IF(OPMWC.EQ.2) OBJBUF(2) = OPMWRD(3)
        IF(OPMWC.EQ.2) OBJBUF(3) = OPMWRD(2)
        OBJWC = OBJWC + OPMWC
        GOTO 7000
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS DECR AND BRANCH INSTRUCTIONS
C       DN,<LABEL>
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
1300    IF(OP1EA.NE.1) GOTO 8500
        OBJBUF(1)=OPSKEL+OP1DA
        I=RFLG
        RFLG=0
        SCAMPT = OPMPT2
        CALL PROCOP(OPMPT2)
        OBJBUF(2)=OPMWRD(2)
        IF(I.EQ.1) RFLG=1
        OBJWC=2
        GOTO 7000
```

(Continued on page 32, column 2)

```
        GOTO 8500
C
1710    OBJWC=2
        OBJBUF(1)=OPSKEL+OP1DA
        OBJBUF(2)=OPMWRD(2)
        GOTO 7000
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS TRAP INSTRUCTION
C        <VECTOR>
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
1800    IF(OP1EA.NE.6) GOTO 8500
        CALL PROCOP(OPMPTR)
        IF(OPMWC.NE.1) GOTO 8500
        IF(OPMWRD(2).GT.16) GOTO 8500
        OBJBUF(1)=OPSKEL+OPMWRD(2)
        GOTO 7000
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS ABCD,SBCD,ADDX,SUBX INSTRUCTIONS
C       DY,DX  -(AY),-(AX)
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
1900    IF(OP1EA.EQ.1.AND.OP2EA.EQ.1) GOTO 1910
        IF(OP1EA.NE.5.OR.OP2EA.NE.5) GOTO 8500
1910    IF(OP1EA.EQ.5) OPSKEL=OPSKEL+8
        OPSKEL=OPSKEL+OP2DA
        OBJBUF(1)=OPSKEL+(OP1DA*'1000)
        GOTO 7000
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS UNLK INSTRUCTION
C       AN
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
2000    IF(OP1EA.NE.2) GOTO 8500
        OBJBUF(1)=OPSKEL+OP1DA
        GOTO 7000

C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS MOVEM,STM,LDM INSTRUCTIONS
C
D       STM <RLIST>,<ADR>  LDM <ADR>,<RLIST>
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
2100    IF(IMODE.EQ.1) GOTO 8500
        IF(IMODE.EQ.3) OPSKEL = OPSKEL.OR.'100
C
C....   TRY PICKING UP A REGISTER OPERAND
C
```

(Continued on page 32, column 3)

```
C
C....   CHECK IF DESTINATION IS LEGAL FOR LDM INSTRUCTION

C....   (AN)+ AND CTL ADR MODES ARE LEGAL
C
        IF ((OP2EA.EQ.3).OR.(OP2EA.EQ.4)) GOTO 2152
        IF (OP2EA.EQ.0) GOTO 2152
        GOTO 8500
C
C....   PROCESS SOURCE OPERAND
C
2152    IF(OP1EA.EQ.0) GOTO 2155
        OBJBUF(1) = OPSKEL.OR.OP1DA.OR.((OP1EA-1)*'10)
        GOTO 2160
C
2155    CALL PROCOP(OPMPTR)
        OBJBUF(1) = OPSKEL .OR. OPMWRD(1)
        OBJBUF(2) = OPMWRD(2)
        IF (OPMWC.EQ.2) OBJBUF(2) = OPMWRD(3)
        IF (OPMWC.EQ.2) OBJBUF(3) = OPMWRD(2)
        OBJWC = OBJWC + OPMWC
C
C....   PROCESS REGISTER LIST
C
2160    OP = OPMPT2
        CALL RLSTDC(OP,DLIST,ALIST)
        IF ((DLIST.EQ.0).AND.(ALIST.EQ.0)) GOTO 8500  ! NO REGISTER LIST!
C
C....   REFORMAT DATA AND ADR BITMAPS FOR LDM INSTRUCTION
C
        CALL BLDMAP(DLIST,ALIST,OBJBUF(OBJWC+1))
        OBJWC = OBJWC + 1
        GOTO 7000
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
C       HANDLE 'NORMAL' SIZE FIELD SPECIFICATIONS
C       USING INFORMATION FROM VARIABLE 'IMODE'
C
C       SIZE FIELD NORMALLY IS IN BITS 6 AND 7 OF
C       INSTRUCTION WITH THE FOLLOWING DEFINITION
C
C            00 = .B  01 = .W  10 = .L
C
C       INSTRUCTIONS WITH IMODE = 0 DEFAULT TO
C       A SIZE OF 'WORD'
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
6000    IF(IMODE.EQ.1) GOTO 7000
        IF(IMODE.EQ.3) OBJBUF(1) = OBJBUF(1).OR.'200
        IF((IMODE.EQ.2).OR.(IMODE.EQ.0)) OBJBUF(1)=OBJBUF(1).OR.'100
C
C ++++++++++++++++++++++++++++++++++++++++++++++++
C
C       PROCESS LABEL FIELD
```

(Continued on next page)

68000 Cross Assembler (XASM)

Listing One

```
C       CURRENT PC VAL IS STORED AS SYMBOL VAL
C
C +++++++++++++++++++++++++++++++++++++++++++++++
C
7000    CALL I4CLR(SYMVAL)
        I=JADD(SYMVAL,PC,SYMVAL)
        OBJWC=OBJWC*2
        I=JICVT(OBJWC,NEWPC)
        OBJWC=OBJWC/2
7005    IF(LABEL(1).EQ.0) RETURN
        CALL SYMTBL(2,SYMVAL,LABEL)
        IF((SYMFLG(STIND).AND.'10).EQ.'10) CALL ERROR(409)
        RETURN
C
C       ERROR DETECTED
C
8500    PRFLG = 0
C
C       IF AN ERROR IS DETECTED, ZERO OBJ BUFFER
C
        DO 8510 I=1,OBJWC
8510    OBJBUF(I) = 0
        IF(PASS.EQ.2) CALL ERROR(406)
        GOTO 7000
        END
        SUBROUTINE RLSTDC(OP,DLIST,ALIST)
        IMPLICIT INTEGER (A-Z)
C
C       THIS SUBROUTINE WILL ATTEMPT TO PROCESS A REGISTER
C       LIST IN THE SOURCE LINE POINTED TO BY 'OP' INTO
C       A PAIR OF WORDS WHICH CAN BE CONVERTED INTO A REGISTER
C       BITMAP FOR THE 'MOVEM' INSTRUCTION
C
        COMMON /SRC   / LNELEN,ISERR,NOCARD,SRCLNE
        LOGICAL*1 SRCLNE(81)
C
C...    INITIALIZE DEFAULT OUTPUT VALUES
C
        DLIST = 0
        ALIST = 0
C
C...    TRY TO FIND A REGISTER TO DECODE
C
10      CALL RDECOD(OP,REGTYP,REGNUM)
        IF (REGTYP.NE.0) GOTO 20
        IF (GRPFLG.EQ.1) GOTO 999
C
C...    A REGISTER WASN'T DETECTED, AND NONE
C...    WAS NECESSARY (REG GROUPS), SO JUST RETURN
C
C       THIS SUBROUTINE RETURNS THE REGISTER TYPE AND NUMBER
C       IF THE NEXT TWO CHARS IN A SOURCE LINE SPECIFY REGISTERS
C
C       REGTYP = 0  NEXT TWO CHRS DON'T SPECIFY A REGISTER
C                1  DATA REGISTER
C                2  ADDRESS REGISTER
C
C       REGNUM =    REGISTER NUMBER (0-7)
C
C       OP     =    OP + 2 UNLESS A REGISTER WASN'T FOUND
C
C
        COMMON /SRC   / LNELEN,ISERR,NOCARD,SRCLNE
        LOGICAL*1 SRCLNE(81)
C
        REGTYP = 0
        IF (SRCLNE(OP).EQ.'101) REGTYP = 2
        IF (SRCLNE(OP).EQ.'104) REGTYP = 1
        IF (REGTYP.EQ.0) RETURN
        OP = OP+1
        IF ((SRCLNE(OP).LT.'60).OR.(SRCLNE(OP).GT.'67)) RETURN
        REGNUM = (SRCLNE(OP).AND.'7)
        OP = OP+1
        RETURN
        END
        SUBROUTINE PROCOP(OP)
C
C       EVALUATE COMPLEX EFFECTIVE ADDRESSES
C
C
C       OUTPUT WORDS:
C
C       OPNFLG  0 IF OPERAND CAN BE USED IN 'QUICK' INSTRUCTIONS
C               1 IF OPERAND CONTAINED A FWD REF SYMBOL
C
C       OPNWC   NUMBER OF BYTES GENERATED (6 MAX)
C
C       OPNWRD  OPERAND WORDS GENERATED
C               FIRST WORD - ADR TYPE
C               NEXT  WORD - OPN DATA <LOW  WORD>
C               NEXT  WORD - OPN DATA <HIGH WORD>
C
        IMPLICIT INTEGER (A-Z)
C
        COMMON /SYMT  / STIND,SYMADR,PC,NOSYM,NEWPC,SYMFLG
C
        COMMON /OPWD  / OPNFLG,OPNWC,OPNWRD
C
        COMMON /OBJOUT/ OBJBUF,OBJWC,LFLG,RFLG,BRFLG
C
        COMMON /LST   / LUNIT,PASS,NAME,NOPAGE,NOLINE,NEFLG,IERCNT
C
      COMMON /PRSE  / OPPTR,MODPTR,OPNPTR,LABEL,CNTPTR
     +,PRFLG,SCANPT,OPCLEN,OPNPT2,IMODE
C
        COMMON /SRC   / LNELEN,ISERR,NOCARD,SRCLNE
        OP = OP+1
        NOCHRS = NOCHRS+1
        IF (NOCHRS.LT.5) GOTO 31
32      IF (SRCLNE(OP).EQ.'47) OP = OP+1
C
C....   CHECK FOR '%' <PC>
C
35      IF(SRCLNE(OP).NE.'52) GOTO 60
        IF(AMD.NE.1) GOTO 40
        I=JADD(SYMVAL,PC,SYMVAL)
        GOTO 25
40      IF(AMD.NE.2) GOTO 9000
        I=JSUB(SYMVAL,PC,SYMVAL)
        GOTO 25
C
C....   CHECK FOR '$' <HEXADECIMAL>
C
60      IF(SRCLNE(OP).NE.'44) GOTO 80
C
C....   HEXADECIMAL LITERAL
C
65      OP=OP+1
        IF(SRCLNE(OP).GE.'60.AND.SRCLNE(OP).LE.'71) GOTO 70
        IF(SRCLNE(OP).GE.'101.AND.SRCLNE(OP).LE.'106) GOTO 75
        GOTO 200
70      NVAL=SRCLNE(OP)-'60
        GOTO 78
75      NVAL=SRCLNE(OP)-'67
78      I=JLSHF(TMPVAL,J4,TMPVAL)
        I=JICVT(NVAL,JADN)
        I=JOR(TMPVAL,JADN,TMPVAL)
        GOTO 65

C
C....   CHECK FOR 0-9 <DECIMAL>
C
80      IF(SRCLNE(OP).LT.'60.OR.SRCLNE(OP).GT.'71) GOTO 100
C
C....   DECIMAL LITERAL
C
85      NVAL=(SRCLNE(OP)-'60)
        I=JMUL(J10,TMPVAL,TMPVAL)
        I=JICVT(NVAL,JADN)
        I=JADD(TMPVAL,JADN,TMPVAL)
        OP=OP+1
        IF(SRCLNE(OP).GE.'60.AND.SRCLNE(OP).LE.'71) GOTO 85
        GOTO 200

C
C....   CHECK FOR A-Z <SYMBOLIC>
C
100     IF(SRCLNE(OP).LT.'101.OR.SRCLNE(OP).GT.'132) GOTO 200
        N=1
        DO 110 OP=OP,OP+7
        IF(SRCLNE(OP).GE.'60.AND.SRCLNE(OP).LE.'71) GOTO 105
        IF(SRCLNE(OP).LT.'101.OR.SRCLNE(OP).GT.'132) GOTO 120
105     TMPSYM(N)=SRCLNE(OP)
110     N=N+1
```

```
C
      RETURN
C
C...  CHECK FOR '/'
C
20    IF(SRCLNE(OP),NE,'57) GOTO 300
C
C...  '/' DETECTED
C
30    IF(GRPFLG,EQ,0) GOTO 200              ! NOT REGISTER GROUP
      IF(STREG,GE,REGNUM) GOTO 999          ! R7-R0 IS ILLEGAL
      IF(STREGT,NE,REGTYP) GOTO 999         ! A0-D0 IS ILLEGAL
C
C...  SET BITS IN REGISTER LIST BITMAP
C
      IF (REGTYP,EQ,2) GOTO 100
      DO 50,I=STREG,REGNUM
50    DLIST = (DLIST,OR,(2**I))
      GOTO 150
100   DO 120,I=STREG,REGNUM
120   ALIST = (ALIST,OR,(2**I))
150   STREG = 0
      REGNUM= 0
      GRPFLG= 0
      OP = OP+1
      GOTO 10
C
C...  ADD AN INDIVIDUAL REGISTER TO LIST
C
200   IF(REGTYP,EQ,1) DLIST = DLIST,OR,(2**REGNUM)
      IF(REGTYP,EQ,2) ALIST = ALIST,OR,(2**REGNUM)
      OP = OP+1
      GOTO 10
C
C....  CHECK FOR '-' OR END OF REGISTER LIST
C
300   IF(SRCLNE(OP),NE,'55) GOTO 30
C
C....  '-' DETECTED, SET UP FOR REG GROUP
C
      STREG = REGNUM
      STREGT= REGTYP
      GRPFLG= 1
      OP = OP+1
      GOTO 10
C
C....  ERROR PROCESSING
C
999   DLIST = 0
      ALIST = 0
      RETURN
      END

      SUBROUTINE RDECOD(OP,REGTYP,REGNUM)
      IMPLICIT INTEGER (A-Z)
```

(Continued on page 34, column 2)

```
C
      COMMON /SYMN  / SYMSYM,SYMLIN
      DIMENSION OBJBUF(40),OPNWRD(3),SYMLIN(512),SYMSYM(4,512)
      LOGICAL*1 TMPSYM(8),SYMFLG(513),NAME(8),SRCLNE(81),LABEL(8)
      INTEGER*4 SYMVAL,TMPVAL,SYMADR(512),J2,J4,J10
      INTEGER*4 PC,NEWPC,J0
C
C....  INITIALIZE I*4 CONSTANTS
C
      J0  = 0
      J2  = 2
      J4  = 4
      J10 = 10
      J256 = 256
C
C....  ZERO OPERAND RESULT BUFFER
C
      DO 10,I=1,3
10    OPNWRD(I) = 0
C
C....  SET PARSE POINTER TO START OF OPN FOR ERROR PROCESSOR
C
      SCANPT = OP
C
C....  DEFAULT IS NON-IMMEDIATE MODE
C....  WITH SUB-OPNS ADDED TO ORIG OPN
C
      IMD   = 0
      AMD   = 1
      OPNWC = 0
      OPNFLG= 0
      STIND = 0
      OPNFLG = 0
      CALL I4CLR(SYMVAL)
      CALL I4CLR(TMPVAL)
C
C....  CHECK FOR ' ' <IMMEDIATE MODE>
C
20    IF(SRCLNE(OP),NE,'43) GOTO 30
      IMD=1
25    OP=OP+1
      CALL I4CLR(TMPVAL)
C
C....  CHECK FOR ASCII LITERAL '
C
30    IF (SRCLNE(OP),NE,'47) GOTO 35
      IMD = 1
      OP = OP+1
      NOCHRS = 0
      CALL I4CLR(TMPVAL)
31    IF (SRCLNE(OP),EQ,'47) GOTO 32
      I=JMUL(J256,TMPVAL,TMPVAL)
      NVAL = SRCLNE(OP)
      I=JICVT(NVAL,JADN)
      I=JADD(TMPVAL,JADN,TMPVAL)
```

(Continued on page 34, column 3)

```
115   IF(SRCLNE(OP),LT,'60) GOTO 120
      IF(SRCLNE(OP),GT,'71,AND,SRCLNE(OP),LT,'101) GOTO 120
      IF(SRCLNE(OP),GT,'132) GOTO 120
      OP=OP+1
      GOTO 115
C
C....  FILL EXTRA CHRS WITH SPACES
C
120   IF(N,GT,8) GOTO 125
      TMPSYM(N) = '40
      N=N+1
      GOTO 120

C
C....  SEARCH SYMBOL TBL
C
125   I=1
      CALL SYMTBL(I,0,TMPSYM)
C
C....  IF SYMLIN LESS THAN CURRENT LINE AND NOT 0
C....  THEN SYMBOL IS DEFINED AND IS NOT A FWD REF
C
      IF((SYMLIN(STIND),LT,NOCARD),AND,(SYMLIN(STIND),NE,0)) GOTO 130
C
C....  CHECK FOR UNDEFINED SYMBOL
C
      IF(SYMLIN(STIND),EQ,0     ) GOTO 150    ! SYMBOL UNDEFINED
C
C....  WE GET TO HERE IF THE SYMBOL IS DEFINED
C....  BUT WASNT AS OF THIS LINE IN THE ASSEMBLY DURING PASS ONE
C
      OPNFLG = 1                              ! SYMBOL WAS FWD REF
C
C....  LABEL HAS BEEN DEFINED
C....  GET VALUE OF LABEL AND PUT IN TMPVAL
C
130   CALL I4CLR(TMPVAL)
      I=JADD(TMPVAL,SYMADR(STIND),TMPVAL)
      GOTO 200
C
C....  GO HERE ON UNDEFINED FIRST AND SECOND PASS SYMBOLS
C
150   IF (PASS,EQ,2) CALL ERROR(407)          ! UNDEFINED SYMBOL !!
C
C     IF THIS IS THE FIRST PASS, THEN THE LENGTH
C     OF ALL OPERANDS OTHER THAN IMMEDIATE BYTE AND WORD
C     ARE FORCED TO TWO WORDS
C
      OPNFLG = 1
      IF((IMD,EQ,1),AND,(IMODE,NE,3)) GOTO 160
      OPNWC = 2
      RETURN
160   OPNWC = 1
      RETURN
C
```

(Continued on next page)

68000 Cross Assembler (XASM)

Listing One

```
C....   PROCESS +,-,*,/,=,!,<<,>>
C
200     IF(AMD.EQ.1)  I=JADD(SYMVAL,TMPVAL,SYMVAL)
        IF(AMD.EQ.2)  I=JSUB(SYMVAL,TMPVAL,SYMVAL)
        IF(AMD.EQ.3)  I=JMUL(SYMVAL,TMPVAL,SYMVAL)
        IF(AMD.EQ.4)  GOTO 205
        IF(AMD.EQ.5)  I=JAND(SYMVAL,TMPVAL,SYMVAL)
        IF(AMD.EQ.6)  I=JOR (SYMVAL,TMPVAL,SYMVAL)
        IF(AMD.EQ.7)  I=JLSHF(SYMVAL,TMPVAL,SYMVAL)
        IF(AMD.EQ.8)  I=JRSHF(SYMVAL,TMPVAL,SYMVAL)
        GOTO 210
C
C....   DIVIDING BY ZERO IS BAD NEWS
C
205     IF(TMPVAL.EQ.0) GOTO 9000
        I=JDIV(SYMVAL,TMPVAL,SYMVAL)
210     AMD=1
C
C....   CHECK FOR +,-,*,/
C
        IF(SRCLNE(OP).NE.'53) GOTO 220
        AMD=1
        GOTO 25
C
220     IF(SRCLNE(OP).NE.'55) GOTO 230
        AMD=2
        GOTO 25
C
230     IF(SRCLNE(OP).NE.'52) GOTO 240
        AMD=3
        GOTO 25
C
240     IF(SRCLNE(OP).NE.'57) GOTO 245
        AMD=4
        GOTO 25
C
245     IF(SRCLNE(OP).NE.'46) GOTO 246
        AMD = 5
        GOTO 25
C
246     IF(SRCLNE(OP).NE.'41) GOTO 247
        AMD = 6
        GOTO 25
C
247     IF(SRCLNE(OP).NE.'74) GOTO 248
        IF(SRCLNE(OP+1).NE.'74) GOTO 9000
        OP = OP+1
        AMD = 7
        GOTO 25
C
        IF(RFLG.NE.0.OR.IMD.EQ.1) GOTO 400
C
C....   IF OPERAND CONTAINED AN EQUATED SYMBOL PROC VAL AS IMMEDIATE
C
        IF((SYMFLG(STIND).AND.'20).EQ.'20) GOTO 400
C
C....   GENERATE PC RELATIVE OFFSET
C
355     OPNWRD(1)='72
        I=JSUB(SYMVAL,J2,SYMVAL)
        I=JSUB(SYMVAL,PC,OPNWRD(2))
        OPNWC=1
        RETURN
C
C....   PROCESS IMMEDIATE DATA
C
400     CALL DBLSGL(SYMVAL,OPNWRD(2),OPNWRD(3))
        IF(IMD.NE.1) GOTO 450
405     OPNWC = 1
        IF(IMODE.EQ.3) OPNWC=2
        OPNWRD(1)='74
        RETURN
C
410     OPNWC=2
        OPNWRD(1)='74
        RETURN
C
C....   PROCESS ABSOLUTE ADR
C....   GENERATE LONG ADR FORM IF INSTR MODE LONG
C
450     IF(OPNFLG.EQ.1) GOTO 460
        IF(OPNWRD(3).NE.0) GOTO 460
        IF(OPNWRD(3).LT.0) GOTO 460
        OPNWC=1
        OPNWRD(1)='70
        RETURN
C
460     OPNWC=2
        OPNWRD(1)='71
        RETURN


C
C....   FATAL ERROR DETECTED

C
9000    OPNWC=7
C
C....   MARK POSITION WHERE ERROR OCCURED
C
        SCANPT = OP
        RETURN
        END


        SUBROUTINE DBLSGL(IN,OUT1,OUT2)
C
C       CONVERT INTEGER*4 TO TWO INTEGER*2 NUMBERS
C
        REG=(SRCLNE(OP+3)-'60)
        GOTO 1070
500     IF(SRCLNE(OP+1).EQ.'101.AND.(SRCLNE(OP+2).GE.'60.AND.
     +SRCLNE(OP+2).LE.'67).AND.SRCLNE(OP+3).EQ.'51)GOTO 510
        RETURN
510     IF(SRCLNE(OP+4).EQ.'53)GOTO 530
        TYP=3
        REG=(SRCLNE(OP+2)-'60)
        GOTO 1075
530     TYP=4
        REG=(SRCLNE(OP+2)-'60)
        GOTO 1070
700     TYP=6
        RETURN
800     TYP=7
        GOTO 1085
900     IF(SRCLNE(OP+2).NE.'122)GOTO 240
        TYP=8
        GOTO 1080
1000    IF(SRCLNE(OP+2).NE.'120)GOTO 240
        TYP=9
        GOTO 1080
1070    IO=SRCLNE(OP+5)
        GOTO 1090
1075    IO=SRCLNE(OP+4)
        GOTO 1090
1080    IO=SRCLNE(OP+3)
        GOTO 1090
1085    IO=SRCLNE(OP+2)
1090    IF(IO.EQ.0.OR.IO.EQ.'40.OR.IO.EQ.'54) RETURN
        IF(TYP.LE.5.AND.TYP.GE.3) GOTO 1110
1100    TYP=0
        RETURN
1110    TYP=10
        RETURN
        END


        SUBROUTINE DECOPC

C
C       LOOKUP OPCODE
C
C       INPUT:OPCODE STARTS AT SRCLNE(OPPTR)
C
        IMPLICIT INTEGER (A-Z)
C
        BYTE LABEL(8),SRCLNE(81),PSUOP3(15),PSUOP4(12)
        BYTE PSUOP5(5),OP4BIG(28),OP4PTY(7),OP3BIG(33)
        BYTE OP3PTY(11),OP3NAM(144),OP3TYP(48),OP4NAM(120)
        BYTE OP4TYP(30),OP5NAM(15)
        DIMENSION OP4OPC(14),OP3OPC(22),OP2OPS(3),OP3OPS(48)
        DIMENSION OP4OPS(30),OP5OPS(3)


C
        COMMON/PRSE/ OPPTR,MODPTR,OPNPTR,LABEL,CMTPTR
     +,PRFLG,SCANPT,OPCLEN,OPNPT2,IMODE
C
        COMMON/SRC/ LNELEN,ISERR,NOCARD,SRCLNE
```

```
C
248     IF(SRCLNE(OP).NE.'76) GOTO 249
        IF(SRCLNE(OP+1).NE.'76) GOTO 9000
        OP = OP+1
        AMD = 8
        GOTO 25
C
249     IF(SRCLNE(OP).NE.'50) GOTO 300
        IF(IMD.EQ.1) GOTO 9000
        IF(SRCLNE(OP+3).NE.'51) GOTO 250
C
C....   A(An)
C
        IF(SRCLNE(OP+1).NE.'101) GOTO 9000
        IF(SRCLNE(OP+2).LT.'60.OR.SRCLNE(OP+2).GT.'67) GOTO 9000
        OPNWC=1
        OPNWRD(1)=(SRCLNE(OP+2)-'60)+'50
        CALL DBLSGL(SYMVAL,OPNWRD(2),OPNWRD(3))
        RETURN
C
C....   A(An,Rn.m)
C
250     CALL DBLSGL(SYMVAL,OPNWRD(2),OPNWRD(3))
        I = ICKVAL(OPNWRD(2))
        IF (I.EQ.0) GOTO 252
        CALL ERROR(408)
        RETURN
C
C....   INDEX OK,.DO THE REST
C
252     OPNWC=1
        IF(SRCLNE(OP+1).NE.'101)GOTO 9000
        IF(SRCLNE(OP+2).LT.'60.OR.SRCLNE(OP+2).GT.'67) GOTO 9000
        OPNWRD(1)=(SRCLNE(OP+2)-'60)+'60
C
C....   CHECK FOR DATA OR ADR REG
C
        IF(SRCLNE(OP+4).EQ.'101.OR.SRCLNE(OP+4).EQ.'104) GOTO 255
        GOTO 9000
255     IF(SRCLNE(OP+4).EQ.'101) OPNWRD(2)=OPNWRD(2)+'100000
        IF(SRCLNE(OP+5).LT.'60.OR.SRCLNE(OP+5).GT.'67) GOTO 9000
        OPNWRD(2)=OPNWRD(2)+((SRCLNE(OP+5)-'60)*'10000)
        IF(SRCLNE(OP+7).EQ.'114) OPNWRD(2)=OPNWRD(2)+'4000
        RETURN
C
C....   CHECK FOR END OF OPERAND
C
300     IF(SRCLNE(OP).EQ.0.OR.SRCLNE(OP).EQ.'40) GOTO 350
        IF(SRCLNE(OP).NE.'54) GOTO 25
C
C....   IF BRANCH INSTRUCTION PROC VAL AS PC REL OFFSET
C
350     IF(BRFLG.EQ.1) GOTO 355
C
C....   PROCESS VAL AS PC REL UNLESS ITS ABS OR IMMEDIATE
```

(Continued on page 36, column 2)

```
        IMPLICIT INTEGER (A-Z)
        DIMENSION IN(2)
        OUT1=IN(1)
        OUT2=IN(2)
        RETURN
        END

        SUBROUTINE EATYP(TYP,REG)
C
C       DETERMINE GENERAL TYPE OF OPERAND
C IN:
C       TYP     = POINTER TO START OF OPERAND
C
C OUT:
C TYP   0       = NOT REGISTER OR IMMEDIATE EA
C       1       = Dn
C       2       = An
C       3       = (An)
C       4       = (An)+
C       5       =-(An)
C       6       = DATA
C       7       = SR
C       8       = CCR
C       9       = USP
C       10      = ERROR DETECTED
C
C REG   REG     0-7
C
        IMPLICIT INTEGER (A-Z)
        COMMON/PRSE/ OPPTR,MODPTR,OPNPTR,LABEL,CMTPTR
     +,PRFLG,SCANPT,OPCLEN,OPNPT2,IMODE
        COMMON/SRC/ LNELEN,ISERR,MOCARD,SRCLNE
        BYTE SRCLNE(81),LABEL(8)
        OP=TYP
100     TYP=0
        IF(SRCLNE(OP).EQ.'43)GOTO 700
        IF(SRCLNE(OP).EQ.'50)GOTO 500
        IF(SRCLNE(OP).EQ.'55.AND.SRCLNE(OP+1).EQ.'50)GOTO 400
        IF(SRCLNE(OP).EQ.'104.OR.SRCLNE(OP).EQ.'101) GOTO 300
210     IF(SRCLNE(OP).EQ.'123.AND.SRCLNE(OP+1).EQ.'122)GOTO 800
220     IF(SRCLNE(OP).EQ.'103.AND.SRCLNE(OP+1).EQ.'103)GOTO 900
        IF(SRCLNE(OP).EQ.'125.AND.SRCLNE(OP+1).EQ.'123)GOTO 1000
240     RETURN
300     IF(SRCLNE(OP+1).LT.'60.AND.SRCLNE(OP+1).GT.'67) RETURN
        IF(SRCLNE(OP).EQ.'101)GOTO 310
        TYP=1
        REG=(SRCLNE(OP+1)-'60)
        GOTO 1085
310     TYP=2
        REG=(SRCLNE(OP+1)-'60)
        GOTO 1085
400     IF(SRCLNE(OP+2).EQ.'101.AND.(SRCLNE(OP+3).GE.'60.AND.
     +SRCLNE(OP+3).LE.'67).AND.SRCLNE(OP+4).EQ.'51)GOTO 410
        RETURN
410     TYP=5
```

(Continued on page 36, column 3)

```
C
        COMMON/OPCPTS/ OPTYP,OPSKEL,OPSK2,OPIDX
C
        DATA PSUOP3/'O','R','G','E','N','D'
     +,'E','Q','U','M','A','M','S','E','T'/
C
        DATA PSUOP4/'R','O','R','G','P','A','G','E'
     +,'L','I','S','T'/
C
        DATA PSUOP5/'N','L','I','S','T'/
C
        DATA OP4BIG/'M','O','V','E','B','C','H','G','B','C','L','R',
     +'B','S','E','T','B','T','S','T','R','O','X','L','R','O','X','R'/
C
        DATA OP4OPC/0,0,'1100,'4100,'700,'4300,'600,'4200
     +,'400,'4000,'160420,'162700,'160020,'162300/
C
        DATA OP4PTY/3,10,10,10,10,8,8/
C
        DATA OP3BIG/'A','D','D','A','S','L','A','S','R','S','U','B'
     +,'A','M','D','C','M','P','E','O','R','L','S','L','L','S','R'
     +,'R','O','L','R','O','R'/
C
        DATA OP3OPC/'150000,'3000,'160400,'160700
     +,'160000,'160300,'110000,'2000
     +,'140000,'1000,'130000,'6000,'130400,'5000
     +,'160410,'161700,'160010,'161300
     +,'160430,'163700,'160030,'163300/
C
        DATA OP3PTY/4,8,8,4,6,5,7,8,8,8,8/
C
        DATA OP2OPS/'100000,'50700,'50300/
C
        DATA OP3NAM/


     +'B','E','Q', 'B','N','E', 'B','P','L', 'B','M','I', 'B','G','T',
     +'B','L','T', 'B','G','E', 'B','L','E', 'B','H','I', 'B','L','S',
     +'B','C','S', 'B','C','C', 'B','V','S', 'B','V','C', 'B','R','A',
     +'B','S','R', 'C','H','K', 'C','L','R', 'E','X','G', 'E','X','T',
     +'J','M','P', 'J','S','R', 'L','D','M', 'L','E','A', 'M','E','G',
     +'N','O','P', 'N','O','T', 'P','E','A', 'R','T','E', 'R','T','R',
     +'R','T','S', 'S','E','Q', 'S','N','E', 'S','P','L', 'S','M','I',
     +'S','G','T', 'S','L','T', 'S','G','E', 'S','L','E', 'S','H','I',
     +'S','L','S', 'S','C','S', 'S','C','C', 'S','T','M', 'S','V','S',
     +'S','V','C', 'T','A','S', 'T','S','T'/
C
        DATA OP3OPS/'63400,'63000,'65000,'65400,'67000,'66400,
     +'66000,'67400,'61000,'61400,'62400,'62000,'64400,'64000,
     +'60000,'60400,'40600,'41000,'140000,
     +'44200,'47300,'47200,'46200,'40700,'42000,'47161,'43000,
     +'44100,'47163,'47167,'47165,'53700,'53300,'55300,'55700,
     +'57300,'56700,'56300,'57700,'51300,'51700,'52700,'52300,
     +'44200,'54700,'54300,'45300,'45000/
C
```

(Continued on next page)

68000 Cross Assembler (XASM)

Listing One

```
      DATA OP3TYP/9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,11,12
     +,14,15,12,12,21,16,12,2,12,12,2,2,2,12,12,12,12,12,12
     +,12,12,12,12,12,12,21,12,12,12,12/
C
      DATA OP4NAM/'A','B','C','D','A','D','D','X','D','B','R','A',
     +'D','B','H','I','D','B','L','S','D','B','C','C','D','B','C','S',
     +'D','B','N','E','D','B','E','Q','D','B','V','C','D','B','V','S',
     +'D','B','P','L','D','B','M','I','D','B','G','E','D','B','L','T',
     +'D','B','G','T','D','B','L','E','D','I','V','S','D','I','V','U',
     +'L','I','N','K','M','U','L','S','M','U','L','U','N','B','C','D',
     +'M','E','G','X','S','B','C','D','S','T','O','P','S','U','B','X',
     +'S','W','A','P','T','R','A','P','U','N','L','K'/
C
      DATA OP4OPS/'140400,'150400,'50710,'51310,'51710,'52310
     +,'52710,'53310,'53710,'54310,'54710,'55310,'55710,'56310
     +,'56710,'57310,'57710,'100700,'100300,'47120,'140700
     +,'140300,'44000,'40000,'100400,'47162,'110400
     +,'44100,'47100,'47130/
C
      DATA OP4TYP/19,19,13,13,13,13,13,13,13,13,
     +13,13,13,13,13,13,13,11,11,17,11,11,12,12,19,2,19
     +,15,18,20/
C
      DATA OP5NAM/'M','O','V','E','M','R','E','S','E','T',
     +'T','R','A','P','V'/
C
      DATA OP5OPS/'44200,'47160,'47166/
C
C     START OF OPCODE PROCESSING
C
      OPTYP=0
      OPSKEL=0
      SCANPT = OPPTR
      IF(OPCLEN.LE.1.OR.OPCLEN.GT.5) RETURN

C
C.... PROCESS OPCODE BY SIZE
C
      GOTO (1000,2000,3000,4000),OPCLEN-1
C
C.... TWO CHR OPCODES
C
1000  IF(SRCLNE(OPPTR).EQ.'104.OR.SRCLNE(OPPTR).EQ.'117)GOTO 1010
      RETURN
1010  IF(SRCLNE(OPPTR).EQ.'117.AND.SRCLNE(OPPTR+1).EQ.'122)GOTO 1020
      IF(SRCLNE(OPPTR+1).EQ.'103)GOTO 1030
      IF(SRCLNE(OPPTR+1).EQ.'123)GOTO 1040
      RETURN
1020  OPTYP=6
      IF(OPTYP.NE.1) GOTO 4010
      OPIDX=11
      OPSKEL=0
      OPSK2=0
      RETURN
4010  CALL OPLOOK(3,5,OP5NAM,OP5OPS,1)
      IF(OPTYP.EQ.0) RETURN
      IF(OPIDX.NE.1) GOTO 4012
      OPTYP=21
      GOTO 4014
4012  OPTYP=2
4014  OPSKEL=OP5OPS(OPIDX)
      OPSK2=0
      RETURN
      END

      SUBROUTINE OPLOOK(ISIZ,ISTEP,ITBL,ITYP,IPSF)
C
C.... LOOK UP OPCODE IN TABLES
C
      IMPLICIT INTEGER (A-Z)
C
      BYTE LABEL(8)
C
      COMMON/PRSE/ OPPTR,MODPTR,OPNPTR,LABEL,CMTPTR
     +,PRFLG,SCANPT,OPCLEN,OPNPT2,IMODE
C
      COMMON /OPCPTS/ OPTYP,OPSKEL,OPSK2,OPIDX
C
      COMMON /SRC/ LNELEN,ISERR,NOCARD,SRCLNE
C
      BYTE SRCLNE(81),ITBL(1),ITYP(1)
C
      IDX=ISIZ * ISTEP
      K=1
      I=1
5     DO 20 IS=1,ISTEP

      IF(SRCLNE(OPPTR+(IS-1)).NE.ITBL(I+(IS-1))) GOTO 10
20    CONTINUE
      OPIDX=K
      IF(IPSF.EQ.0) GOTO 30
      OPTYP=ITYP(K)
      RETURN
10    I=I+ISTEP
      K=K+1
      IF(I.GE.IDX) RETURN
      GOTO 5
30    OPTYP=1
      RETURN
      END

      SUBROUTINE PARSE
C
C     PARSE INCOMING SOURCE LINE
C
C IN:
C     SRCLNE  = LINE TO BE PARSED
C
C     LABELS HAVE TO START WITH A-Z
C
      IF(SRCLNE(1).GE.'101.AND.SRCLNE(1).LE.90) GOTO 30
20    CALL ERROR(202)
      RETURN
C
30    DO 40 SCANPT=1,8
      IF (SRCLNE(SCANPT).GE.48.AND.SRCLNE(SCANPT).LE.57) GOTO 35
      IF(SRCLNE(SCANPT).LT.65.OR.SRCLNE(SCANPT).GT.90) GOTO 45
35    LABEL(SCANPT)=SRCLNE(SCANPT)
40    CONTINUE
45    IF(SCANPT.GE.4) GOTO 50
      IF(LABEL(1).EQ.'101.OR.LABEL(1).EQ.'104) GOTO 46
      IF(LABEL(1).EQ.'123) GOTO 47
      IF(LABEL(1).EQ.'103) GOTO 48
      IF(LABEL(1).NE.'125) GOTO 50
      IF(LABEL(2).EQ.'123.AND.LABEL(3).EQ.'120) GOTO 49
      GOTO 50
C
46    IF(SCANPT.GT.3) GOTO 50
      IF(LABEL(2).GE.'60.AND.LABEL(2).LE.'67) GOTO 49
      GOTO 50
C
47    IF(LABEL(2).EQ.'120.OR.LABEL(2).EQ.'122) GOTO 49
      GOTO 50
C
48    IF(LABEL(2).EQ.'103.AND.LABEL(3).EQ.'122) GOTO 49
      GOTO 50
C

49    PRFLG=0
      CALL ERROR(204)
      RETURN
C
50    IF(SRCLNE(SCANPT).EQ.'40.OR.SRCLNE(SCANPT).EQ.'72) GOTO 60
      PRFLG=0
      CALL ERROR(205)
      RETURN
C
60    SCANPT=SCANPT+1
      PRFLG=1
62    IF(SRCLNE(SCANPT).NE.'40) GOTO 70
      SCANPT=SCANPT+1
      GOTO 62
C
70    IF(SRCLNE(SCANPT).EQ.0) GOTO 12
      OPPTR=SCANPT
      DO 80 I=1,5
      IF(SRCLNE(SCANPT).LT.65.OR.SRCLNE(SCANPT).GT.90) GOTO 90
      SCANPT=SCANPT+1
80    CONTINUE
C
C.... LENGTH OF OPCODE IS ONE LESS THAN  SCANNED
90    OPCLEN=I-1
C
C.... CHECK FOR END OF LINE
      IF(SRCLNE(SCANPT).EQ.0) RETURN
```

```
        OPIDX=0
        OPSKEL='100000
        OPSK2=0
        RETURN
1030    OPTYP=1
        OPIDX=1
        OPSKEL=0
        OPSK2=0
        RETURN
1040    OPTYP=1
        OPIDX=2
        OPSKEL=0
        OPSK2=0
        RETURN
C
C....   THREE CHR OPCODES
C
2000    CALL OPLOOK(5,3,PSUOP3,OP3TYP,0)
        IF(OPTYP.NE.1) GOTO 2010
        OPIDX=OPIDX+2
        OPSKEL=0
        OPSK2=0
        RETURN
2010    CALL OPLOOK(11,3,OP3BIG,OP3PTY,1)
        IF(OPTYP.EQ.0) GOTO 2020
        OPSKEL=OP3OPC((OPIDX*2)-1)
        OPSK2=OP3OPC(OPIDX*2)
        RETURN
2020    CALL OPLOOK(48,3,OP3NAM,OP3TYP,1)
        OPSKEL=OP3OPS(OPIDX)
        OPSK2=0
        RETURN
C
C....   FOUR CHAR OPCODES
C
3000    CALL OPLOOK(3,4,PSUOP4,OP3NAM,0)
        IF(OPTYP.NE.1) GOTO 3010
        OPIDX=OPIDX+7
        OPSKEL=0
        OPSK2=0
        RETURN
3010    CALL OPLOOK(7,4,OP4BIG,OP4PTY,1)
        IF(OPTYP.EQ.0) GOTO 3020
        OPSKEL=OP4OPC((OPIDX*2)-1)
        OPSK2=OP4OPC(OPIDX*2)
        RETURN
3020    CALL OPLOOK(30,4,OP4NAM,OP4TYP,1)
        IF(OPTYP.EQ.0)  RETURN
        OPSKEL=OP4OPS(OPIDX)
        OPSK2=0
        RETURN
C
C....   FIVE CHAR OPCODES
C
4000    CALL OPLOOK(1,5,PSUOP5,OP3TYP,0)
```

(Continued on page 38, column 2)

```
C       LNELEN  = LENGTH OF SOURCE LINE
C OUT:
C       LABEL   = LABEL FIELD (LABEL(0)=0 IF NO LABEL)
C       OPPTR   = POINTER TO OPCODE FIELD
C       OPCLEN  = LENGTH  OF OPCODE FIELD NOT INCLUDING MODE
C       MODPTR  = POINTER TO MODE FIELD
C       IMODE   = 0 NO MODE FIELD
C               = 1 .B
C               = 2 .W
C               = 3 .L
C               = 4 .S
C       OPNPTR  = POINTER TO FIRST OPERAND
C       OPNPT2  = POINTER TO SECND OPERAND
C       CMTPTR  = POINTER TO COMMENT FIELD
C       PRFLG   = PARSE FLAG - ZERO IF ERROR DETECTED
C
        IMPLICIT INTEGER (A-Z)
        COMMON /SRC/ LNELEN,ISERR,NOCARD,SRCLNE
        COMMON/PRSE/ OPPTR,MODPTR,OPNPTR,LABEL,CMTPTR
     +,PRFLG,SCANPT,OPCLEN,OPNPT2,IMODE
        BYTE SRCLNE(81),IC,LABEL(8),MODTBL(4)
        DATA MODTBL/'102,'127,'114,'123/
        IPF   = 0
        IMODE = 0
        PRFLG = 1
        LABEL(1)=0
C
C       INITALIZE LABEL ARRAY TO ALL SPACES
C
        DO 10 I=2,8
        LABEL(I)=32
10      CONTINUE
        SCANPT = 1
        OPCLEN = 0
        OPPTR = 0
        OPNPTR = 0
        OPNPT2 = 0
        MODPTR = 0
        CMTPTR = 0
C
C       IF NULL LINE IGNORE IT
C
        IF(LNELEN.NE.1) GOTO 15
12      PRFLG=0
        RETURN
C
C       CHECK FOR A LINE OF COMMENTS
C
15      IF(SRCLNE(1).NE.'52) GOTO 16
        CMTPTR=1
        RETURN
C
C       SEE IF LABEL PRESENT
C
16      IF(SRCLNE(SCANPT).EQ.'40) GOTO 60
```

(Continued on page 38, column 3)

```
C
C....   CHECK FOR SPACE
        IF(SRCLNE(SCANPT).EQ.'40) GOTO 112
C
C....   CHECK FOR xxx.x
        IF(SRCLNE(SCANPT).EQ.'56) GOTO 100
C
C....   IF NOT EOL,SPC,OR PERIOD GEN ERROR
95      OPPTR=0
        PRFLG=0
        CALL ERROR(207)
        RETURN
C
C....   CHECK FOR .B .W .L .S
C....   POINT TO SIZE SUBFIELD
100     SCANPT=SCANPT+1
C
C....   SCAN FOR VALID SIZE
        DO 102,IMODE = 1,4
102     IF(MODTBL(IMODE).EQ.SRCLNE(SCANPT))GOTO 105
C
C....   IF NOT IN TABLE IT'S INVALID
        IMODE = 0
        GOTO 95
C
C....   SAVE POSITION OF MODE FIELD
105     MODPTR=SCANPT
C
C....   CHECK FOR SPACE AFTER OPCODE
110     SCANPT=SCANPT+1
        IF(SRCLNE(SCANPT).NE.'40) GOTO 95
C
C....   PARSE FIRST OPERAND IF THERE
112     SCANPT=SCANPT+1
        IC=SRCLNE(SCANPT)
        IF(IC.EQ. 0 ) RETURN
        IF(IC.EQ.'40) GOTO 112
        IF(IC.EQ.'44.OR.IC.EQ.'52) GOTO 114
        IF ((IC.EQ.'50).OR.(IC.EQ.'47)) GOTO 114
        IF(IC.EQ.'55.OR.IC.EQ.'43) GOTO 114
        IF(IC.GE.'60.AND.IC.LE.'71) GOTO 114
        IF(IC.LT.'101.OR.IC.GT.'132) GOTO 95
C
C....   SAVE START OF FIRST OPERAND
C
114     OPNPTR=SCANPT
        IF ((SRCLNE(SCANPT).NE.'47).AND.(SRCLNE(SCANPT+1).NE.'47))
     +    GOTO 116
        IF (SRCLNE(SCANPT+1).EQ.'47) SCANPT = SCANPT + 1
115     SCANPT = SCANPT+1
        IF(SRCLNE(SCANPT).EQ.0) GOTO 118
        IF(SRCLNE(SCANPT).NE.'47) GOTO 115
116     SCANPT=SCANPT+1
        IC=SRCLNE(SCANPT)
        IF((IC.EQ.0).OR.(IC.EQ.'40)) GOTO 118
```

(Continued on next page)

68000 Cross Assembler (XASM)

Listing One

```
      IF(IC.EQ.'50) IPF=1
      IF(IC.EQ.'51) IPF=0
      IF(IC.EQ.'54.AND.IPF.EQ.0) GOTO 120
      GOTO 116
118   OPNPT2=0
      IF (IC.NE.'40) RETURN
119   SCANPT = SCANPT+1
      IF (SRCLNE(SCANPT).EQ.'40) GOTO 119
      CMTPTR = SCANPT
      RETURN
C
C....  SAVE START OF SECOND OPERAND
C
120   OPNPT2=SCANPT+1
125   SCANPT = SCANPT + 1
      IF (SRCLNE(SCANPT).EQ.'40) GOTO 130
      IF (SRCLNE(SCANPT).EQ.0  ) RETURN
      IF (SRCLNE(SCANPT).NE.'47) GOTO 125
127   SCANPT = SCANPT + 1
      IF (SRCLNE(SCANPT).EQ.0  ) RETURN
      IF (SRCLNE(SCANPT).NE.'47) GOTO 127
      GOTO 125
130   SCANPT = SCANPT + 1
      IF (SRCLNE(SCANPT).EQ.'40) GOTO 130

      CMTPTR = SCANPT
      RETURN
      END
```

68000 XASM Subroutines

Listing Two

```
      .NLIST  TTM,BEX
      .TITLE  MC68000 CROSS ASSEMBLER SUPPORT
;
; ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;
; ASSEMBLY LANGUAGE SUBROUTINES FOR MC68000 CROSS-ASSEMBLER.
; FORTRAN LINKAGE TO THESE ROUTINES IS AS FOLLOWS:
;       1. RETURN VIA 'RTS PC'.
;       2. R5 POINTS TO PARAMETER LIST WITH FOLLOWING:
;             A. NUMBER OF PARAMETERS.
;             B. ADDRESS OF FIRST PARAMETER
;             C. ADDRESS OF SECOND PARAMETER, ETC.
;       3. FUNCTION SUBROUTINES (INTEGER) RETURN VALUE IN R0.
;
; ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;
      TST     (R5)+           ;SKIP COUNT ARG
      MOV     (R5)+,R4        ;ADDR OF OPERAND 1
      MOV     (R5)+,R3        ;ADDR OF OPERAND 2
      MOV     (R5)+,R1        ;ADDR OF RESULT
      MOV     (R4)+,R2        ;GET LOW ORDER OP 1
      ADD     (R3)+,R2        ;ADD IN LOW ORDER OP 2
      MOV     @R4,R5          ;GET HIGH ORDER OP 1
      ADC     R5              ;ADD IN CARRY
      BVS     1$
      ADD     @R3,R5          ;ADD IN HIGH ORDER OP 2
      BVC     2$
1$:   CLR     R2              ;SET RESULT TO 0
      CLR     R5
      MOV     -2,R0           ;INDICATE OVERFLOW
2$:   MOV     R2,(R1)+        ;STORE RESULT
      MOV     R5,@R1
      RTS     PC
      .PAGE
;
;       INTEGER*4 SUBTRACTION
;
JSUB::  CLR     R0              ;INITIALIZE FOR NORMAL EXIT
      TST     (R5)+           ;SKIP COUNT ARG
      MOV     (R5)+,R4        ;ADDR OF OPERAND 1
      MOV     (R5)+,R3        ;ADDR OF OPERAND 2
      MOV     (R5)+,R1        ;ADDR OF RESULT
      MOV     (R4)+,R2        ;GET LOW ORDER OP 1
      SUB     (R3)+,R2        ;SUBTRACT LOW ORDER OP 2
      MOV     @R4,R5          ;GET HIGH ORDER RESULT
      SBC     R5              ;SUBTRACT CARRY
      BVS     1$
      SUB     @R3,R5          ;SUBTRACT HIGH ORDER OP 2
      BVC     2$
1$:   CLR     R2              ;SET RESULT TO 0
      CLR     R5
      MOV     -2,R0           ;INDICATE OVERFLOW
2$:   MOV     R2,(R1)+        ;STORE RESULT
      MOV     R5,@R1
      RTS     PC
      .PAGE
;
;       INTEGER*4 MULTIPLICATION
;
JMUL::  TST     (R5)+           ;SKIP COUNT ARG
      CLR     -(SP)           ;INIT SIGN FLAG
      MOV     (R5)+,R4        ;ADDR OF OPERAND 1
      MOV     (R4)+,R1        ;GET LOW ORDER OP 1
      MOV     @R4,R3          ;GET HIGH ORDER OP 1
      BPL     1$              ;BRNCH IF POSITIVE
      NEG     R3              ;TAKE ABSOLUTE VALUE
      NEG     R1
      SBC     R3
      INC     @SP             ;AND SET SIGN FLAG
1$:   MOV     (R5)+,R4        ;ADDR OF OPERAND 2
      MOV     (R4)+,R0        ;LOW ORDER OF OP 2
      MOV     @R4,R2          ;HIGH ORDER OF OP 2
      BPL     2$              ;BRAANCH IF POSITIVE
      INC     @SP             ;SET SIGN FLAG
      BEQ     ZDIV            ;BRANCH IF ZERO DIVIDE
2$:   MOV     (R4)+,R2        ;LOW ORDER OF NUM
      MOV     @R4,R3          ;HIGH ORDER OF NUM
      BPL     3$              ;BRANCH IF POSITIVE
      ADD     40000,@SP       ;SET SIGN FLAG
      NEG     R3              ;AND TAKE ABSOLUTE VALUE
      NEG     R2
      SBC     R3
3$:   CLR     R5              ;QUOTIENT ENDS UP IN R3:R2
      CLR     R4              ;REMAINDER ENDS UP IN R5:R4
4$:   ROL     R4              ;EXPOSE NEW BIT OF NUMERATOR
      ROL     R5
      CMP     R1,R5           ;DOES DENOM FIT?
      BHI     6$              ;BRANCH IF NOT, C=0
      BNE     5$              ;BRANCH IF YES
      CMP     R0,R4           ;HIGH PARTS SAME, CHECK LOW
      BHI     6$              ;BRANCH IF NOT, C=0
5$:   SUB     R0,R4           ;SUBTRACT DENOM FROM REMAINDER
      SBC     R5
      SUB     R1,R5
      SEC                     ;INDICATE NEW QUOTIENT BIT
6$:   ROL     R2              ;SHIFT IN NEW BIT OF QUOTIENT
      ROL     R3
      DECB    @SP             ;CHECK LOOP COUNT
      BGT     4$              ;BRANCH TO LOOP
ENDCOD: ASL   @SP             ;PUT QUOTIENT RESULT SIGN IN V
      BVC     1$              ;BRANCH IF TO BE POSITIVE
      NEG     R3              ;NEGATE QUOTIENT
      NEG     R2
      SBC     R3
1$:   TST     (SP)+           ;GET REMAINDERS SIGN
      BPL     2$              ;BRANCH IF REMAINDER TO BE POSITIVE
      NEG     R5              ;NEGATE REMAINDER
      NEG     R4
      SBC     R5
2$:   MOV     (SP)+,R0        ;GET ARG LIST POINTER AGAIN
      MOV     (R0)+,R1        ;ADDR OF QUOTIENT RESULT
      CMPB    3,(SP)+         ;CHECK NUMBER OF ARGUMENTS
      BEQ     3$              ;BRANCH IF ONLY 3 PASSED
      MOV     @R0,R0          ;GET ADDR REMAINDER RESULT
      MOV     R4,(R0)+        ;STORE LOW
      MOV     R5,@R0          ;STORE HIGH
3$:   MOV     R2,(R1)+        ;STORE LOW QUOTIENT RESULT
      MOV     R3,@R1          ;STORE HIGH
      TST     (SP)+
      RTS     PC
ZDIV:   CLR     R4              ;SET REMAINDER TO 0
      CLR     R5
      CLR     R2              ;SET QUOTIENT TO 0
      CLR     R3
      MOV     -3,6(SP)        ;SET ZDIV ERROR INDICATOR
      BR      ENDCOD          ;GO SET RESULTS
      .PAGE
;
;       INTEGER*4 LOGICAL AND
;
JAND::  TST     (R5)+           ;SKIP COUNT ARGUMENT
      MOV     (R5)+,R0        ;ADR OF NUMBER
```

```
; ** I4CLR **
;
; CLEAR LOW AND HIGH WORDS OF DOUBLE PRECISION VARIABLE
; ADR OF LOW WORD PASSED VIA (R5)
;
I4CLR:: TST (R5)+
        MOV R0,-(SP)
        MOV (R5),R0
        CLR (R0)+
        CLR (R0)
        MOV (SP)+,R0
        RTS PC
;
; ** GETBIT **
;
; SUBROUTINE TO CONVERT 4 LSB OF A INTEGER*2 VARIABLE TO A HEX
; ASCII DIGIT.  THE INTEGER IS IN THE FIRST PARAMETER AND THE
; HEX DIGIT IS RETURNED IN THE SECOND PARAMETER (BYTE).  THE
; INTEGER IS SHIFTED RIGHT BY 4 BEFORE RETURNING.
;
CNVTBL: .ASCII  '0123456789ABCDEF'
GETBIT::        MOV     R0,-(SP)                ;
        MOV     @2(R5),R0               ;GET WORD
        BIC      177760,R0              ;MASK OUT 4 LSB
        MOVB    CNVTBL(R0),@4(R5)       ;CONVERT AND RETURN
        MOV     @2(R5),R0               ;SHIFT INPUT INTEGER
        ASH      -4,R0                  ; RIGHT BY 4.
        MOV     R0,@2(R5)               ;
        MOV     (SP)+,R0                ;
        RTS     PC                      ;RETURN
        .PAGE
;
; ** ICKVAL **
;
; INTEGER FUNCTION TO CHK IF I*2 VARIABLE IS IN THE RANGE
; -64,63. IF IT IS A VALUE OF 0 IS RETURNED, OTHERWISE
; A VALUE OF ONE IS RETURNED

;
ICKVAL::        MOV     @2(R5),R0
        BIT      177600,R0
        BEQ     10$
        BIC      177,R0
        CMP     R0, 177600
        BEQ     10$
        MOV      1,R0
        RTS     PC
10$:    CLR     R0
        RTS     PC
        .PAGE
;
;       THE FOLLOWING INTEGER*4 ARITHMETIC ROUTINES ARE USED INSTEAD
;       OF F4P CALLS TO PERMIT THE CROSS ASSEMBLER TO RUN WITHOUT
;       MODIFICATIONS ON EITHER RT-11 OR RSX-11 SYSTEMS
;
JADD::  CLR     R0              ;INITIALIZE FOR NORMAL EXIT
```

(Continued on page 40, column 2)

```
        NEG     R2              ;AND TAKE ABSOLUTE VALUE
        NEG     R0
        SBC     R2
2$:     BEQ     3$              ;BRANCH IF HIGH PART OF OP 2 IS 0
        TST     R3              ;IF WASN'T OP 2 MUST BE OP 1
        BNE     OVRFL           ;BRANCH IF RESULT WILL BE TOO BIG
        MOV     R0,R4           ;OTHERWISE SWITCH OPS
        MOV     R1,R0
        MOV     R4,R1
        MOV     R2,R3
        CLR     R2
3$:     CLR     R4              ;RESULT WILL END UP IN R2:R4
4$:     ROR     R0              ;SHIFT BIT OUT OF MULTIPLIER
        BCC     5$              ;BRANCH IF 0
        ADD     R1,R4           ;ADD LOW PARTS TOGETHER
        ADC     R2              ;ADD IN CARRY
        BVS     OVRFL           ;BRANCH IF OVERFLOW
        ADD     R3,R2           ;ADD HIGH PARTS TOGETHER
        BVS     OVRFL           ;BRANCH IF OVERFLOW
        TST     R0              ;ANY MORE OF MULTIPLIER LEFT?
5$:     BEQ     DONE            ;BRANCH IF FINISHED
        ASL     R1              ;SHIFT MULTIPLICAND LEFT
        ROL     R3
        BVC     4$              ;LOOP
OVRFL:  MOV      -2,R0          ;SET OVEFLOW INDICATOR
        CLR     R2              ;SET RESULT TO 0
        CLR     R4
DONE:   ROR     (SP)+           ;GET RESULT SIGN INTO C
        BCC     1$              ;BRANCH IF TO BE POSITIVE
        NEG     R2              ;NEGATE RESULT
        NEG     R4
        SBC     R2
1$:     MOV     @R5,R1          ;ADDR OF RESULT
        MOV     R4,(R1)+        ;STORE LOW ORDER
        MOV     R2,@R1          ;STORE HIGH ORDER
        RTS     PC
        .PAGE
;
;       INTEGER*4 DIVISION
;
JDIV::  CLR     -(SP)           ;SET SUCCCESSFUL EXIT FLAG
        MOV     (R5)+,-(SP)     ;SAVE ARG COUNT
        MOV     (R5)+,R4        ;ADDR OF NUMERATOR
        MOV     (R5)+,R1        ;ADDR OF DENOMINATOR
        MOV     R5,-(SP)        ;SAVE ADDR OF RESULT
        MOV      33.,-(SP)      ;SET SHIFT COUNTER
        MOV     (R1)+,R0        ;GET LOW ORDER DENOM
        MOV     @R1,R1          ;GET HIGH ORDER DENOM
        BPL     1$              ;BRANCH IF POSITIVE
        NEG     R1              ;TAKE ABSOLUTE VALUE
        NEG     R0
        SBC     R1
        ADD      100000,@SP     ;AND SET SIGN FLAG
1$:     BNE     2$              ;BRANCH IF DENOM CAN'T BE 0
        TST     R0              ;MAKE SURE LOW DENOM ISN'T 0
```

(Continued on page 40, column 3)

```
        MOV     (R5)+,R1        ;ADR OF NUMBER TO BE ANDED
        MOV     (R5)+,R2        ;ADR OF DESTINATION
        MOV     (R0)+,R3        ;GET VAL OF HIGH PART OF NUMBER
        MOV     (R1)+,R4        ;GET VAL OF HIGH PART OF MASK
        COM     R4              ;INVERT MASK BITS
        BIC     R4,R3           ;AND 'EM
        MOV     R3,(R2)+        ;STORE RESULT IN DESTINATION
        MOV     (R0),R3         ;GET VAL OF LOW PART OF NUMBER
        MOV     (R1),R4         ;GET VAL OF LOW PART OF MASK
        COM     R4              ;INVERT MASK BITS
        BIC     R4,R3           ;AND 'EM
        MOV     R3,(R2)         ;STORE RESULT IN DESTINATION
        CLR     R0              ;CLR VAL OF FUNCTION
        RTS     PC
        .PAGE
;
;       INTEGER*4 INCLUSIVE OR
;
JOR::   TST     (R5)+           ;SKIP OVER ARGUMENT COUNT
        MOV     (R5)+,R0        ;ADR OF NUMBER
        MOV     (R5)+,R1        ;ADR OF NUMBER TO BE OR'ED
        MOV     (R5)+,R2        ;ADR OF DESTINATION
        MOV     (R0)+,R3        ;GET VAL OF HIGH PART OF NUMBER
        MOV     (R1)+,R4        ;GET VAL OF HIGH PART OF OR MASK
        BIS     R4,R3           ;OR 'EM
        MOV     R3,(R2)+        ;STORE RESULT IN DESTINATION
        MOV     (R0),R3         ;GET VAL OF LOW PART OF NUMBER
        MOV     (R1),R4         ;GET VAL OF LOW PART OF OR MASK
        BIS     R4,R3           ;OR 'EM
        MOV     R3,(R2)         ;STORE RESULT IN DESTINATION
        CLR     R0              ;STD FUNCTION RETURN VAL
        RTS     PC
        .PAGE
;
; ** JLSHF **
;
;       INTEGER*4 LEFT SHIFT (UNSIGNED)
;
JLSHF:: TST     (R5)+           ;SKIP OVER ARGUMENT COUNT
        MOV     (R5),R0         ;SRC ADR
        MOV     @2(R5),R1       ;NUMBER OF BITS TO SHIFT IT
        MOV     4(R5),R2        ;DEST ADR
        MOV     (R0)+, (R2)     ;MOVE SRC TO DEST
        MOV     (R0) ,2(R2)
        TST     R1
        BEQ     99$             ;SHIFTING 0 BITS IS SILLY
        CMP     R1, 40
        BHI     99$             ;CAN'T SHIFT MORE THAN 32 BITS
1$:     CLC                     ;MAKE SURE CARRY IS CLEAR
        ROL     (R2)            ;SHIFT LO HALF 1 BIT
        ROL     2(R2)           ;SHIFT HI WRD
3$:     DEC     R1              ;DECR  BITS TO SHIFT
        BNE     1$              ;LOOP IF NON ZERO
;
99$:    CLR     R0
        RTS     PC
```

(Continued on next page)

68000 XASM Subroutines

Listing Two

```
        .PAGE
;
; ** JRSHF **
;
;       INTEGER*4 RIGHT SHIFT (UNSIGNED)
;
JRSHF:: TST     (R5)+           ;SKIP OVER ARGUMENT COUNT
        MOV     (R5),R0         ;ADR OF NUMBER TO SHIFT
        MOV     @2(R5),R1       ;NUMBER OF BITS TO SHIFT IT
        MOV     4(R5),R2        ;DESTINATION ADDRESS
        MOV     (R0)+, (R2)     ;MOVE SRC TO DEST
        MOV     (R0) ,2(R2)
        TST     R1
        BEQ     99$             ;SHIFTING 0 BITS IS SILLY
        CMP     R1, 40
        BHI     99$             ;CAN'T SHIFT MORE THAN 32 BITS
1$:     CLC                     ;MAKE SURE CARRY IS CLEAR
        ROR     2(R2)           ;SHIFT HI HALF 1 BIT
        ROR     (R2)            ;SHIFT LO WRD
        DEC     R1              ;DECR  BITS TO SHIFT
        BNE     1$              ;LOOP IF NON ZERO
;
99$:    CLR     R0
        RTS     PC
        .PAGE
;
; ** JICMP **
;
;       COMPARE A 32 BIT SIGNED NUMBER WITH A 16 BIT SIGNED NUMBER
;       FUNCTION RETURNS A VALUE OF ZERO IF THE NUMBERS ARE EQUIVALENT
;
```

(Continued on column 2)

```
JICMP:: TST     (R5)+
        MOV     (R5)+,R0        ;ADR OF I*4 VALUE
        MOV     @(R5)+,R1       ;VALUE OF I*2 NUMBER
        TST     R1              ;SEE IF VAL NEGATIVE
        BPL     1$              ;
        CMP     (R0), 177777    ;SEE IF NUMBER NEGATIVE
        BNE     99$             ;NUMBER NOT NEGATIVE
        BR      2$
1$:     TST     (R0)            ;IF I*2 NUMBER NOT NEG THEN HI WORD
        BNE     99$             ;MUST BE ZERO
2$:     CMP     2(R0),R1        ;CMP THE TWO NUMBERS
        BNE     99$
        CLR     R0
        RTS     PC              ;NUMBERS ARE EQUAL
99$:    MOV      1,R0           ;NUMBERS NOT EQUAL
        RTS     PC
        .PAGE
;
; ** JMOV **
;
;       INTEGER*4 MOVE
;
JMOV::  CLR     R0              ;INITIALIZE FOR NORMAL EXIT
        TST     (R5)+           ;SKIP COUNT ARG
        MOV     (R5)+,R4        ;ADDR OF OPERAND 1
        MOV     (R5)+,R1        ;ADDR OF OPERAND 2
        MOV     (R4)+,(R1)+     ;MOVE LOW PART
        MOV     @R4,@R1         ;MOVE HIGH PART
        RTS     PC
        .PAGE
;
; ** BLDMAP (DLIST,ALIST,OUTPUT) **
;
;       CREATE A REGISTER BITMAP FROM A DATA AND ADR REG MAP
;
BLDMAP:: TST    (R5)+           ;SKIP OVER ARG COUNT
        MOV     @(R5)+,R0       ;GET DATA BITMAP
```

(Continued on column 3)

```
        BIC      177400,R0      ;MAKE SURE NO EXTRA BITS SET
        MOV     @(R5)+,R1       ;GET ADR BITMAP
        BIC      177400,R1      ;MAKE SURE NO EXTRA BITS SET
        SWAB    R1              ;MOVE ADR REG BITS UP WHERE THEY BELONG
        BIS     R0,R1           ;OR IN THE D-REG BITS
        MOV     R1,@(R5)        ;SAVE THE RESULT IN THE DESTINATION
        RTS     PC              ;AND RETURN
        .PAGE
;
; ** JICVT (I*2,I*4 RESULT) **
;
;       I*2 TO I*4 CONVERSION (SIGNED)
;
JICVT:: TST     (R5)+           ;SKIP COUNT ARG
        CLR     R2              ;INIT SIGN EXTEND WORD
        MOV     @(R5)+,R0       ;GET INTEGER VALUE
        BPL     1$              ;BRANCH IF POSITIVE
        DEC     R2              ;SET SIGN EXTEND WORD
1$:     MOV     @R5,R1          ;ADDR OF RESULT
        MOV     R0,(R1)+        ;STORE LOW ORDER
        MOV     R2,@R1          ;STORE HIGH ORDER
        CLR     R0              ;INITIALIZE FOR NORMAL EXIT
        RTS     PC
        .END
```

68000 XASM Demo

Listing Three

```
        NAM     68K_DEMO
CONSTANT EQU    $1000   A CONSTANT
NAME    DC.L    'THE ','QUIC','K BR','OWN ','FOX '
        DC.W    0
LENGTH  EQU     *-NAME
*
*       CROSS-ASSEMBLER DEMO PROGRAMME
*
LABEL:  MOVE.L  #$10000,A0      HEXADECIMAL SOURCE OPN
        CMP.L   #10000,A0       DECIMAL RADIX
        MOVE.L  #$F<<16,D0      EXAMPLE OF SHIFT LEFT
*
*       DO SOMETHING USEFUL, LIKE MOVE A TABLE
*
        MOVE.L  #TABLE1,A0      GET ADR OF TABLE1 INTO A0
        MOVE.L  #TABLE2,A1      GET ADR OF TABLE2 => A1
LOOP:   MOVE    (A1)+,(A0)+     MOVE IT (DEFAULT IS .W)
        CMP     #TABLE2,A0      DONE?
        BNE     LOOP            NOPE
        JMP     LABEL           AND LOOP FOREVER
TABLE1: DS.W    10
TABLE2  DC.W    $10,$20,$30     COLONS ARE OPTIONAL
        END
```

(Continued on top half of page 43)

WILD Listing

(Listing continued from page 16, text begins on page 12)

```
/*****************************************************************
*                                                               *
*       wtest.c -- example program for wildcard processing      *
*                                                               *
*       component of WILD --- package to provide wildcard       *
*       processing within a Unix(VII) program.                  *
*                                                               *
*       by Anthony Skjellum.                                    *
*       Copyright 1982 (c) All rights reserved.                 *
*                                                               *
*       May be used for all non-commercial purposes             *
*       and freely distributed for non-commercial               *
*       purposes as long as all copyright notices               *
*       are retained.                                           *
*                                                               *
*****************************************************************/


#include <stdio.h>
#include <wild.h>

main(argc,argv)
int argc;
char **argv;
{


        WILD check;
        int i = 1;
        char input[100]; /* will get this input */

        printf("Enter file specification --> ");
        scanf("%s",input); /* get the spec. */

        check = wpopen(input); /* open... */
        printf("\n\n"); /* add some space for readability */



        do
        {
                int error = wnext(&check); /* get next */

                if(error) break; /* done... */

                printf("%u\t\t%s\n",i++,check.W_fnam); /* print */


        } while(1);

        printf("\n\ndone...\n\n");
        wclose(&check); /* close */

}
```

End Listing

```
 1 00000000                             NAM     68K_DEMO
 2 00000000                     CONSTANT EQU     $1000                  A CONSTANT
 3 00000000 5448 4520 5155 4943 4B20 NAME    DC.L    'THE ','QUIC','K BR','OWN ','FOX '
 4 00000014 0000                         DC.W    0
 5 00000016                     LENGTH   EQU     *-NAME
 6                                       *
 7                                       *       CROSS-ASSEMBLER DEMO PROGRAMME
 8                                       *
 9 00000016 207C 0001 0000      LABEL    MOVE.L  #$10000,A0             HEXADECIMAL SOURCE OPN
10 0000001C B1FC 0000 2710              CMP.L   #10000,A0              DECIMAL RADIX
11 00000022 203C 000F 0000              MOVE.L  #$F<<16,D0             EXAMPLE OF SHIFT LEFT
12                                      *
13                                      *       DO SOMETHING USEFUL, LIKE MOVE A TABLE
14                                      *
15 00000028 207C 0000 0040              MOVE.L  #TABLE1,A0             GET ADR OF TABLE1 INTO A0
16 0000002E 227C 0000 0054              MOVE.L  #TABLE2,A1             GET ADR OF TABLE2 => A1
17 00000034 30D9                LOOP    MOVE    (A1)+,(A0)+            MOVE IT (DEFAULT IS .W)
18 00000036 B0FC 0054                   CMP     #TABLE2,A0             DONE?
19 0000003A 66F8                        BNE     LOOP                   NOPE
20 0000003C 4EF8 0016                   JMP     LABEL                  AND LOOP FOREVER
21 00000040                     TABLE1  DS.W    10
22 00000054 0010 0020 0030      TABLE2  DC.W    $10,$20,$30            COLONS ARE OPTIONAL
23                                      END


CONSTANT 00001000 EQ    LABEL    00000016      LENGTH   00000016 EQ    LOOP     00000034      NAME     00000000
TABLE1   00000040       TABLE2   00000054

  7 SYMBOLS ,   0 ERRORS DETECTED

000000 5448 4520 5155 4943 4B20 4252 4F57 4E20
000010 464F 5820 0000 207C 0001 0000 B1FC 0000
000020 2710 203C 000F 0000 207C 0000 0040 227C
000030 0000 0054 30D9 B0FC 0054 66F8 4EF8 0016
000054 0010 0020 0030
```

End Listing

Memory Management in BASIC

The allocation of memory is made at the beginning of the program by the use of a DIM statement. This is called static memory allocation because the size of the dimensioned variable cannot be changed during the execution of the program. There are times, however, when one needs dynamic memory allocation; that is the ability to create and delete arrays as the program is being executed. The primary reason why one would need such capability is when two or more arrays are required, but not simultaneously, and there is not enough memory to represent them separately.

It is possible to allocate the maximum amount of memory that will be needed by using a single dimension statement and dynamically, within the program, operate upon two or more arrays as if they had been defined at the beginning of the program. There are two methods of accessing arrays: column major order and row major order. Information stored according to column major order would be A(1, 1), A(2, 1), A(1, 2), A(2, 2), A(1, 3), and A(2, 3). Note that the leftmost subscript changes first. Row major order is the method of storing data in memory a row at a time. An array A(2, 3) would be stored in memory as A(1, 1), A(1, 2), A(1, 3), A(2, 1), A(2, 2), and A(2, 3). Again note the subscript; the rightmost subscript changes first.

Management Methods

The equations which will permit one to define the arrays are,

$$B + ((J - 1) * R + I - 1)) \quad (1)$$

$$B + ((I - 1) * C + J - 1)) \quad (2)$$

Equation (1) is for column major order and equation (2) is for row major order. The definitions of the variables in the equations are:

- B represents the beginning address of the array
- R represents the number of rows
- C represents the number of columns
- I is the row subscript
- J is the column subscript

by Henry Seymour

Henry A. Seymour, Senior Designer – Computer Systems, Martin Marietta Aerospace, P.O. Box 29304, New Orleans, LA 70189.

In order to demonstrate this capability I will define an array, read data into it, and manipulate the data by using equations (1) and (2). Pictorially, the memory layout is shown in Figure 1.

Original	Column Major Order	Row Major Order
X(1)	Y(1, 1)	Z(1, 1)
X(2)	Y(2, 1)	Z(1, 2)
X(3)	Y(1, 2)	Z(1, 3)
X(4)	Y(2, 2)	Z(2, 1)
X(5)	Y(1, 3)	Z(2, 2)
X(6)	Y(2, 3)	Z(2, 3)
X(7)		
X(8)		
X(9)		

Figure 1.
Memory layout.

As shown in Figure 2, an array X has been created and filled with the values 1 to 9.

```
110 DIM X(9)
120 MAT READ X
130 DATA 1,2,3,4,5,6,7,8,9
140 LET B = 1
150 LET R = 2
160 FOR J = 1 TO 3
170 FOR I = 1 TO 2
180 PRINT X(B + ((J - 1) * R + I - 1)),
    I, J
190 NEXT I
200 NEXT J
210 LET B = 2
220 LET C = 3
230 FOR I = 1 TO 2
240 FOR J = 1 TO 3
250 PRINT X(B + ((I - 1) * C + J - 1))
    I, J
260 NEXT J
270 NEXT I
```

Figure 2.
Program demonstrating memory management.

The variable B in statement 140, which represents the beginning address, has been set to one. This implies that the pseudo-array Y that is about to be emulated will begin at the same address as the array X. That is X(1) and Y(1, 1) are equivalent. The print statement is being used as an example in order to reference the pseudo-array. The output from print statement 180 is,

1	1	1
2	2	1
3	1	2
4	2	2
5	1	3
6	2	3

which represents a two-dimensional array Y(2, 3) stored in column major order. Graphically, it would be represented as,

	J		
I	1	3	5
	2	4	6

The code beginning at statement 210 equates X(2) and Z(1, 1). The print statement 250 produces the output,

2	1	1
3	1	2
4	1	3
5	2	1
6	2	2
7	2	3

which represents a two-dimensional array Z(2,3) stored in row major order. Graphically, it would be represented as,

	J		
I	2	3	4
	5	6	7

Summary

The programmer has the flexibility, power, and ease of use of two equations which allows the dynamic definition of memory. By changing the value of B one may equate a hypothetical array any place along the continuum of the original array and by using one of the equations it is possible to access the data in either column or row major order. Also, multiple arrays may be defined simultaneously, thereby allowing the coexistence of multiple dynamic arrays. This will permit some programs to run in a restricted memory size that could not otherwise run. The maximum amount of memory required must be dimensioned and is allocated during the time of the execution of the program. ▶▶J

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

CP/M-86 vs. PC-DOS Benchmarks

Now that both Digital Research CP/M-86 and Microsoft PC-DOS are readily available for the IBM Personal Computer, we finally have the opportunity to compare the performance of these two operating systems in the same hardware environment. Dave Cortesi's technical analysis published in *DDJ* July 1982 pointed out that although the systems appear very similar to the casual user, they use drastically different allocation schemes to manage disk files. This has surprisingly large effects on the speed of disk operations. Here are some example timings, run on an IBM PC with 128 Kbytes RAM and two double-sided, double-density, 5¼-inch diskette drives.

	PC-DOS	CP/M-86
Batch copy of three data files; total data transferred=101 Kbytes	26 sec.	86 sec.
Sequentially read a 10 Kbyte file in 128-byte blocks	4.6 sec.	3.5 sec.
Perform 100 semi-random reads of 128 byte records, 10 Kbyte file (identical record number sequence used on both systems)	13.5 sec.	32.0 sec.

Now for the Bad News . . .

PC-DOS gets a large measure of its snappy performance from a resident File Allocation Table (FAT) that completely describes the blocks in use on the diskette. The operating system reads this table into memory when a file is opened, and does not refer back to the diskette copy of the table thereafter. If a file changes size during processing, the modified directory entry and the FAT are written back to the disk when the file is closed. This process apparently does not have any safeguards.

Consider the following situation: You are using a program, such as an editor, that executes with a file open. You accidentally change diskettes before leaving the editor. When the editor receives the command to exit, it naturally flushes its buffers and reissues a request to the operating system to close the working file. PC-DOS unilaterally writes the directory and File Allocation Table from the previous diskette onto the new diskette, *making it completely unusable without any warning!* This disaster has struck me twice already, and I believe it is bound to happen to everyone sooner or later. Such lack of concern for data integrity on the part of PC-DOS's designers is very worrisome. In contrast, CP/M-86 always checks the identity of a floppy disk before modifying directory information; if it finds a mismatch it will simply mark the disk as "Read-Only" and return an error message.

The PC-DOS disk storage scheme can also lead to inefficient use of disk space under certain circumstances. If your program creates a random access file and only writes a few scattered records, CP/M-86 will assign just enough disk blocks to hold the actual data, while PC-DOS will also allocate the "missing" empty records whether they are ever used or not.

More on the Intel 8087

The price of the 8087 arithmetic coprocessor chip continues to fall, now listing for $237 in single unit quantities from your friendly local distributor. If you're still wondering what all the fuss is about, Figure 1 shows some benchmarks run on the IBM PC that will make the power of the 8087 obvious.

Microsoft Fortran does not have a double precision (64-bit) floating-point data type, so a true comparison could not be carried out.

You might wonder what the effect will be on your BASIC programs if you plug an 8087 into your PC. Unfortunately, you won't see any improvement. Even though Microsoft had some 40 Kbytes of ROM to use for their runtime subroutines, they couldn't find the space for logic that would check for the existence of an 8087 in the system and exploit it if available. There are rumors that when PC-DOS version 2.0 is released, it will require a new ROM set that will include drivers for the numeric coprocessor.

In the meantime, South Coast Data Sciences of Costa Mesa, California, has announced 8087 support libraries for the Fortran, COBOL, Pascal, and BASIC compilers. Their press releases imply that you must also buy a matched pair of 8088 and 8087 chips (from SCDS, of course) and discard the 8088 that is currently in your PC. This is not true. Although some of the early engineering samples of the 8087 did have problems with timing and overheating, the current production 8087-3 will work correctly with any current production 8088 at 5 MHz.

16-Bit Toolbox, Installment #3

This month's subroutine is an integer square root using Newton's Method. Listing #1 is written in Intel 8086 assembly language and Listing #2 in Forth. Please note that the result is always rounded *down*, not to the nearest integer. This algorithm is the basis for the square root subroutine in most compiler run-time libraries. Interesting discussions can be found in Donald Knuth's *Art of Computer Programming* series or Kernighan and Plauger's *Elements of Programming Style*.

When used on a floating-point number, a test for convergence is typically used after every iteration. When working on integers of limited magnitude, it is actually faster to just perform a fixed number of iterations that will be adequate for the largest possible argument, since the test for convergence requires an extra multiplication and division operation. This square root routine will be used as a building block for a fast integer Chebyshev approximation of sine and cosine in next month's column. DDJ

Operation (5000 iterations)	Microsoft Fortran 32-bit floating point	Forth with 8087 64-bit floating point
multiply	5.1 sec.	3.3 sec.
divide	20.9 sec.	3.4 sec.
sine	6 min. 59 sec.	6.4 sec.
square root	5 min. 45 sec.	2.3 sec.

Figure 1.

Newton's Square Root

Listing One

```
                                title   'Integer Square Root'
                                pagewidth 79
                                cseg
                        ;
                        ; Integer Square Root by Newton's Method
                        ; Ray Duncan, August 15, 1982
                        ;
                        ; Call with:      AX = Argument
                        ; Returns:        AX = Square root
                        ;                 Other registers preserved
                        ;
0000 3D0100             sqrt:   cmp     ax,1    ;exit if argument
0003 7F01       0006            jg      sqrt1   ;less than two
0005 C3                         ret
0006 53                 sqrt1:  push    bx      ;save registers
0007 51                         push    cx
0008 52                         push    dx
0009 8BD8                       mov     bx,ax   ;initial approximation:
000B D1FB                       sar     bx,1    ;K <- X/2
000D B91400                     mov     cx,20   ;set loop counter
0010 50                 sqrt2:  push    ax      ;save copy of X
0011 99                         cwd             ;extend to 32 bits
0012 F7FB                       idiv    bx      ;next approximation:
0014 03D8                       add     bx,ax   ;K <- (K+(X/K))/2
0016 D1FB                       sar     bx,1
0018 58                         pop     ax      ;restore copy of X
0019 E2F5       0010            loop    sqrt2   ;keep approximating
001B 8BC3                       mov     ax,bx   ;copy answer to AX
001D 5A                         pop     dx      ;then restore registers
001E 59                         pop     cx
001F 5B                         pop     bx
0020 C3                         ret             ;return to caller
                        ;

END OF ASSEMBLY.  NUMBER OF ERRORS:   0.  USE FACTOR:  0%
```

Listing 2

```
Screen # 55
  0 ( Integer square root by Newton's method                04/08/82 )
  1
  2 20 CONSTANT SQRT-#-ITER
  3
  4 ( n   ---  square-root )
  5 : SQRT          DUP 0<
  6                 IF ."  Illegal argument" 0 ERROR
  7                 THEN DUP
  8                 IF   DUP 2/
  9                      SQRT-#-ITER 0
 10                      DO
 11                            2DUP / + 2/
 12                      LOOP
 13                      SWAP DROP
 14                THEN ;
 15
```

End Listing

SOFTWARE REVIEW

JRT Pascal: A Cheap Pascal Compiler
Published by JRT Systems
$29.95
Circle No. 487 on Reader Service Card.
Reviewed by Dave Cortesi

"If it works, it's a real bargain." That was the typical reaction of everyone who heard that a full Pascal compiler could be had from JRT Systems of San Francisco for $29.95. I am a strong advocate of cheap software; few things gall me so much as software prices that are set as high as the market will bear, then justified by claims of the high cost of support (which never materializes), and protected by paranoid locking schemes. I was prepared to give JRT Pascal a glowing review and to become an avid booster of JRT Systems.

It really hurts to have to report that JRT Pascal is not a bargain. It implements a highly non-standard version of the language for no very good reason. Considering the price, that could be forgiven if it worked reliably. It does not.

Let me address the issue of language standards first. The original description of Pascal (*Pascal User Manual and Report*, by Kathlene Jensen and Niklaus Wirth, Springer-Verlag, 1975) is quite definite on what the Pascal language consists of. It leaves a few academic loose ends, but by and large an implementor couldn't ask for a clearer, better-organized description of a language.

Wirth's definition omitted features that other people thought (rightly) were necessary in a general-purpose language. Almost every Pascal implementor has felt it necessary to add support for (1) varying-length strings of characters, (2) separate compilation of program modules with later linking of modules, and (3) random access to files. Since there was no standard in these matters, every implementor did them in his own way. As a result, you just can't write a transportable Pascal program that uses any of these three features. But a program that adheres strictly to the language as described by Jensen and Wirth *should* be transportable.

JRT Pascal supports strings, separate compilation, and random I/O in ways that are, as expected, incompatible with any other Pascal (any other that is listed in that excellent reference, *The Pascal Handbook*, Jacques Tiberghien, Sybex, 1981, at any rate). But JRT Pascal not only supplements the standard language, it deviates from it in many ways. It allows the plus-sign to mean concatenation of strings, for example. It does not enforce Pascal strong typing; structured variables of differing lengths may be compared, and a structured type can be assigned to another that is shorter or longer. It rejects the keyword "packed" (it should be ignored if not implemented) and doesn't recognize the standard type "text" as meaning "file of char." Compiler directives, rather than being hidden in specially-formatted comments, are elevated to the status of language keywords. That reserves a number of words that are not reserved words in standard Pascal. A program that declares a variable named "list" will fail to compile, for example.

A particularly irritating deviation is the implementation of sequential I/O. Wirth's scheme for I/O is elegant: a variable declared as a "file of *type*" simply becomes a pointer to an object of that type. The standard procedure "get" makes the file variable point to the next item of that type in the file. It's the high-level-language equivalent of memory-mapped I/O in hardware. JRT Pascal ignores Wirth and requires the programmer to use the "read" and "readln" procedures for files of every type. That makes Pascal I/O look more like Fortran or BASIC, which some might find comforting. It also makes JRT Pascal programs incompatible with standard Pascal compilers.

Well, suppose that you don't mind writing non-portable programs and are willing to take the compiler as just another Pascal-like applications language. How does it work? JRT Pascal is a partial compiler. You prepare a source program with your choice of text editors. The command JRTPAS2 invokes the compiler. It reads the source, writes a listing and any error messages to the CP/M console, and writes a file of type .INT to disk. The .INT file contains code for a virtual machine that is implemented by a run-time interpreter named EXEC. That program loads the intermediate-code file and executes it.

For reasons that will become clear, I wasn't able to make any performance tests of JRT-compiled programs. The compiler itself is a JRT-compiled program; it runs slowly compared to native-code programs but reasonably fast for an intermediate-code implementation. It compiles at a rate of about three lines per second.

When it compiles at all, that is. What really damns this product is that the compiler is very fragile; it falls apart at a touch. Lest you think I exaggerate, here is a brief log of my use of JRTPAS2. All the programs I tried it on were working ones that compiled correctly under a more expensive product. Program One caused JRT to bomb after about thirty lines with the run-time message "Array index too small" and EXEC's dump of a failing program's stack. Program Two provoked the compile error "Type identifier expected" after a line "type page=record. . .". That message was followed by four identical messages about an "Invalid dynamic storage address," and then the compiler aborted with "Array index too large." Later I found that "page" is a reserved word in JRT Pascal – but this is not the way that I'd choose to notify the programmer of that fact.

In Program Three, the "program" statement followed a comment block which began with "[" but ended with "*)". The compiler said it couldn't find the "program" statement. Then it entered a loop in which "invalid dynamic storage address" messages alternated with odd-sounding disk activity. I hastily mashed the system reset button, after which my system wouldn't re-boot until I powered it down and up again.

That scared me, so I left off compiling my own programs and turned to a set of the programs from the Jensen and Wirth book that I had on disk. I tried to compile seven of these; two of them (programs 11.3 and 11.5) compiled correctly and ran. Program 11.5 takes keyboard input one character at a time. It appeared from its behavior that when a program asks for a single character, EXEC reads an entire line from the keyboard, gives the program the first byte, and throws the rest of the line away.

The other five failed to compile. One had a variable (not a constant) named "list"; the compiler reacted with "Error in constant" and then aborted with run-time errors. Another used the perfectly standard declaration "type infile=text" which produced the same uninformative results. Another passed a function as a parameter of a procedure. JRT Pascal doesn't support that, but doesn't diagnose it where it appears in the formal parameter list. It diagnoses an "Unknown variable name" when the parameter is

(Continued on page 53)

BOOK REVIEW

Teaching Your Computer To Talk
by Edward R. Teja
Published by TAB Books
$15.95 hardback, $8.95 paperback
Reviewed by Joseph B. Rothstein

Voice I/O – the use of spoken words to interact with computers – is one of the industry's "hot buttons" at the moment. Billion-dollar markets are forecast for speech-based products, both on the input and output sides. Business executives demand alternatives to the keyboard if they are to take computers into their inner offices and their hearts. Futurists predict a new wave of technology, based on friendly interaction with intelligent, speaking and listening systems. And the military, of course, is watching and waiting while doing some experimenting of its own.

For the newcomer to the field, the array of predictions, applications and devices may seem bewildering. Speech synthesis and voice recognition have been near the top of researchers' "wish lists" since the earliest days of computing, but it seems like the wish has become reality practically overnight. It was just a couple of years ago that Texas Instruments introduced Speak and Spell; now, suddenly, there are talking bathroom scales, talking alarm clocks and talking calculators, besides a host of other talking toys and learning aids.

Despite the giant strides made in recent years, the field is still wide open to the type of garage R & D which brought us microcomputers in the first place. A number of writers for the popular computer journals have recently sounded the death knell for individuals working nights and weekends in their basements and garages, citing big corporations and research institutions which have all but cornered the market on microcomputer research and development. But as one window closes, another opens. Voice technology may provide the means for hobbyists and experimenters to recapture the early, heady days of the Altair, Microsoft BASIC and CP/M – while staking out a share of the profits, as those earlier entrepreneurs did.

A new book from TAB, publishers of numerous tutorials on computers, ham radios and similar topics, may help place the subject in perspective and offer an opportunity for the computer hobbyist to enter this exciting and potentially rewarding field. Entitled *Teaching Your Computer to Talk*, by Edward R. Teja, it covers the historical context, scientific basis and application of speech synthesis and voice recognition technology.

The author – an Associate Editor of *EDN Magazine* – softpedals speculation and prediction, concentrating instead on current applications, available hardware and microcomputer interfacing techniques. It's a considered and well-focused approach, recommended for those who have at least some background in digital electronics. The emphasis is on hardware, with numerous schematics and circuit diagrams included. Software, however, is given scant treatment – only one listing for the Z-80 is offered – and while this may minimize dependence on a particular computer, it leaves some significant gaps which will have to be filled by the experimenter.

That caveat aside, the tutorial is a thorough and down-to-earth approach to a subject which too often carries an aura of sci-fi. A general survey and history of voice recognition and synthesis, while brief, establishes a context for the discussion which follows. Similarly, a section covering the physics and biomechanics of speech finds a solid middle ground between boring those familiar with the subject and confusing those new to the field.

The bulk of the text – and its greatest strength – is its analysis of a wide range of speech synthesis, voice recognition and related hardware. The range of the discussion runs the gamut from such products as the Kurtzweil Computer Products Reading Machine costing many thousands of dollars (which can "read" aloud from printed material and is designed to aid the visually handicapped) to the $395 Votrax Audio 1 speech synthesis module for the Pet computer, targeted at the hobby and experimenter market, to a simple $70 interface from Percom Data Company which connects a TI Speak & Spell to a Radio Shack TRS-80 computer.

The product evaluations are concise and direct, stressing likely applications, interfacing techniques and cost. Each product is discussed in a level of detail appropriate to its usefulness to the experimenter. Prices are given for each product considered at length, and the names and addresses of product manufacturers are included in a six-page appendix that is handy for ordering or reference.

Both the phoneme synthesis and linear predictive coding (LPC) approaches are covered, with attention to the full range of speech related devices available, from chips – for example, the $100 (quantity 100) TMS5200 LPC chip from Texas Instruments, combining dozens of subsystems in a single integrated circuit – to board-level modules and stand-alone devices such as the Model 4700, a fully-equipped recognition and response system from Centigram Corporation, priced at $3500. In this fashion, the experimenter may choose the level of complexity and challenge appropriate to his or her interests, expertise and budget.

If there is any real shortcoming to this text, it's that it is almost *too* timely. In this fast-changing field, it's likely that new hardware introductions, new prices and new applications will change the outlook for speech technology faster than any book can keep up with. This is a minor point, though, and one which it would be a pleasure to raise about many other such books. On the whole, the survey of products is as complete and thorough as any experimenter has a right to expect.

The implications of this technology are staggering, and anyone interested in the social impacts of computers would do well to consider how this emerging field will affect our business, personal and institutional lives in the coming years. Other books and magazine articles cover this aspect of voice I/O more thoroughly than the nuts-and-bolts approach taken here. While there is tantalizing mention of talking robots, pilot guidance systems and interactive telephone directories, these barely scratch the surface of applications currently under development, much less those in the imagination stage. But this is just as well, for there are many sources of such surveys and speculations, some of them cited in the text and too-brief bibliography.

Similarly, those with no hardware background, and those without previous exposure to voice-driven devices, may do best to look elsewhere first before tackling the projects outlined in this tutorial and reference. But the eager experimenter, undaunted by schematics and anxious to plunge into an application field still in its infancy, would be hard-pressed to find a more comprehensive and valuable starting point than *Teaching Your Computer To Talk*. ▶▶J

OF INTEREST

by Michael Wiesenberg

Hardware 'n' Things

PConnection, from Microperipheral, is an IBM PC modem card, a direct-connect, Bell 103/113 compatible modem with autodial (Touch Tone or pulse), that answers automatically in both originate and answer modes. The card fits inside the PC cabinet, and costs $350. **Reader Service No. 307.**

The Osborne 1 can now use **Corvus Systems'** network and mass storage disk systems, in configurations of 6, 11, and 20 megabytes. The Corvus Multiplexer local area network permits up to seven Osborne users to share up to 80 megabytes, in addition to printers and other peripherals. The disk systems for Osborne 1 start at $3195, including interface card that plugs right into the IEEE-488 port, and software. **Reader Service No. 317.**

Turn your TRS-80 model I or III into an IBM PC emulator with **μMERLIN** from Micro Projects Engineering Co. The self-contained unit has an 8088, 8K of EPROM, and 64K of dynamic RAM (expandable to 256K). The TRS-80 becomes a slave I/O processor for access to CRT, keyboard, disks, and I/O ports. Optional interval timer and priority interrupt controller chips allow later expansion to multitasking operating systems. μMERLIN comes with CP/M-86; MS-DOS is coming early next year. This unit provides software compatibility with the PC, and costs $1195 and up. **Reader Service No. 327.**

If you write a lot, as I do, you go through a lot of printer ribbons. **Mac Inker** from Computer Friends is an automatic printer ribbon reinker that permits virtually endless reuse of ribbons. Available with many different colors of ink and cartridge loading stations for all printers, Mac Inker costs $54.95. **Reader Service No. 337.**

Applications 'n' Such

A recent major study of CP/M word processors compared **Spellbinder**, from Lexisoft, Inc., with twelve other similar products for micros. In the study's more than 100 categories, Spellbinder received over twice as many positive ratings as the average, and 10% more than its closest contenders, many of whose "extra" features could be obtained only for extra cost, but are standard on Spellbinder. Although Spellbinder has been around for awhile, it now comes with a new office and data management package. Spellbinder is a full-screen editor that uses all the features of most letter-quality precision printers, typographic-quality printers, and dot-matrix printers. It uses special function keys on most terminals, offers help messages for most operations, manages files larger than memory, and executes command files. Their standard 20,000-word spelling checker, **Spellcheck**, that can be expanded to the capacity of a complete disk, for example, can be operated as a system overlay with a command that returns control to Spellbinder. In addition to all the standard capabilities that you expect from a word processor, Spellbinder scrolls horizontally to accommodate lines of up to 160 characters. It word wraps onto variable and independently set left margins. It deletes or moves with one keystroke by character, word, line, multiple line, sentence, or paragraph. It has three kinds of hyphenation, breaking (splits at any of the hyphens of "father-in-law"), nonbreaking (always keeps something like "Section A-1" together), and "soft" (appears only when that part of the word falls at the end of a line, as in "hyphen-ation"). Spellbinder prints to the screen or to disk instead of a printer if you wish, or right from memory without having to save to disk. You have complete dynamic control over the printer, with variable line spacing adjustable to 1/48th inch, and true proportional spacing. And if you don't like the widths Spellbinder assigns to characters, you can change them. You can also change character spacing within lines. You can number pages at top or bottom, and use headers or footers or both, each with separate formats for even and odd pages. You can assign commands or any repetitive sets of keystrokes into a specific single function-keystroke. The office management package includes forms creation and fill-in. The Spellbinder disk comes with macros for mail merge, cue sorting, alphabetizing, and Zip Code sorting. You can print in multiple columns, each justified, and print in patterns, such as a document printed in the shape of a Christmas tree or a large circle. Math functions include column and row addition, decimal tabbing, columnar editing, and fixed-point arithmetic. All this goes for $495. **Reader Service No. 347.**

Can you run a program with over 200K of code on an 8-bit CP/M system? **MEGA/CPM** from SysteMathica extends the nominal 64K address space by swapping program segments in and out of main memory with overlay management transparent to the user (and, they claim, to the CPU). MEGA/CPM requires no hardware or software modifications, suffers no speed degradation with external ROM/RAM, and, when disks are used, its speed is determined by the transfer rate of the disk controller. The CP/M 2.2 version costs $100; MEGA/CPM on HP series 80 is $150; MEGA/ISIS for ISIS-II costs $190. **Reader Service No. 357.**

Softworks C is the first available higher-level language for the TRS-80 Model 16, Radio Shack's 68000-based entry into the 16-bit market. Softworks Limited claims theirs is a complete implementation of the C language. It is based on the Whitesmith's C compiler, and costs $950. Softworks also offers cross-compilers to develop Model 16 C programs on other systems. **Reader Service No. 367.**

HP's HP Plus program distributes and supports third-party software, including the **Condor** database management system for the HP 125. This DBMS uses the 125's screen-labeled "softkeys" and editing keys. It writes files that can be read into the HPWORD/125 word-processing system (HP's implementation of Lexisoft's Spellbinder), and used by HPGRAPHICS/125 graphics software. Condor uses menus to create databases, make entries and queries, manipulate data, and generate reports. It produces databases of up to 32K entries, each with up to 127 fields. Condor Series 20-1 creates databases in minutes, extracts reports soon thereafter, and costs $295. Series 20-2 handles multiple files, relating data among them, and projects, posts, and combines information from all files, and costs $595. The upgrade from 20-1 to 20-2 costs $350. **Reader Service No. 377.**

Interested in C, Ada, and Lisp? **SuperSoft** offers a free 24-page brochure describing their implementations of those languages, and other goodies, including a virtual memory electronic spreadsheet called Scratchpad that never runs out of memory, and Disk-Edit, a screen-oriented disk editor that simultaneously displays programs in hexadecimal and ASCII. **Reader Service No. 387.**

Altos offers a 70-page software guide, costing $5, for its ACS8000 8-bit and ACS8600 16-bit micros, including hundreds of packages running under CP/M, CP/M-86, MP/M, MP/M-86, and XENIX/UNIX. **Reader Service No. 397.**

Two free catalogs of educational

software from many publishers for Apple, PET, TRS-80, and Atari are offered by Queue, Inc. One is for kindergarten through ninth grade, and the other for high school and college. Specify computer and grade level when writing. **Reader Service No. 407.**

An upgrade of **AmperSoft** from Micro-SPARC, Inc., for Apple II with RAM card, moves the DOS onto the card, thus providing an extra 10K of usable program memory. It also offers new functions: &Print for BASIC's Print Using; &Sort, a machine-language sort routine; &Store/Recall, storage and retrieval of numeric arrays as binary files that take up half the space of, and are accessed up to 16 times faster than, standard text files; &Matrix, for matrix operations at machine-language speed; &FRE, for clearing memory of character strings in seconds; and &CLEAR, to clear memory of arrays. The disk also includes hooks to the PLE, FID, MUFFIN, RENUMBER, and COPYA utilities to be used with the relocated DOS. This copyable disk costs $49.95 at Apple dealers, or add $1.50 postage and handling to order from Micro-SPARC. **Reader Service No. 417.**

WSP FORTH for the IBM PC, from World Wide Software Publishers, is based on the fig-Forth standard. It comes with its own editor and utilities, and costs $75. **Reader Service No. 427.**

Miscellany 'n' Etc.

The **C Users' Group**, a non-profit organization, has over 1200 members. You can join for $10 a year ($20 foreign), and get six issues of the newsletter and updates to registered purchasers of the BDS C compiler. CUG offers library disks in most formats for $4 to $8 each. **Reader Service No. 437.**

SQ (Syntax Quarterly) is a magazine for users of the Timex-Sinclair (née ZX81). Published by SYNTAX, the Timex-Sinclair newsletter, an SQ subscription is $15 for four issues, or one year of SQ plus twelve monthly issues of SYNTAX for $39. **Reader Service No. 447.**

The Software Author SIG on CompuServe is for freelance programmers and writers in the microcomputer field. It maintains an electronic bulletin board, conference facilities, and market report databases. Run by Saturday Software, the service is free. A CompuServe subscriber issues the command G PCS-117 from any prompt, or selects option 13 from the SIG's menu on PCS-50.

Each month, we get a wonderful publication called **Privacy Journal** that addresses the erosion of the average citizen's privacy in the computer age. Usually it contains nothing of direct interest to *DDJ* readers, but the July issue's main article was about skinnydipping, "Right to Be Let Alone: Nude Swimming." In a sidebar called "The Naked Programmer," Lee Baxandall, coordinator of the Free Beach Documentation Center (great name!), an organization that wants to open recreation sites to nude use, states that a "disproportionately large number of computer programmers are skinnydippers. It's no coincidence that . . . many nude beaches [are] located near Silicon Valley," because programmers and other DP folks practice "diverse life styles and are more tolerant of others' practices." The fact that his organization, The Naturists, has a great many computerists among its members has caused him to consider advertising in electronics industry trade publications. If you want to join programmers and other Naturists, write to Box 132, Oshkosh, WI 94901. The July issue of *Privacy Journal* also contained an article warning about the ease of government surveillance of its citizens through credit cards. When credit cards are used for travel and entertainment, monitoring and curtailing the movements of free citizens becomes easier. A subscription to *Privacy Journal* is $79 ($95 overseas). They also publish *Compilation of State and Federal Privacy Laws*, an 80-page book, at $16. **Reader Service No. 457.**

The **Personal Computer Products Buyer's Guide** from Starware brings together software and hardware for the IBM PC from a host of manufacturers. Starware offers a 10% discount for orders from them of any product in the guide, a six-month free hardware repair warranty, software support, and free phone advice. They also pay all shipping charges. Starware will send one issue free; a one-year subscription (12 issues) is $8. Any order also gets a free subscription. (They have a 20% royalty arrangement for software developers.) **Reader Service No. 467.**

Among other projects, the **San Diego Computer Society** is trying to establish a format on 5¼-inch diskettes for interchange of data between otherwise incompatible machines and operating systems. The SDCS has eight special-interest subgroups (SIGs), and members seem to own the spectrum of computer systems and are interested in all languages and operating systems. They put out a classy newsletter, *Personal Systems*. Membership is $15 for the first year, and $10 thereafter. **Reader Service No. 477.**

DDJ

(Continued from page 48)

referenced. Two other programs put the compiler into a loop from which it hadn't emerged after several minutes.

JRT Pascal claims a lot of attractive features, but there's no point in going into them here. It carries a very attractive pricetag for its claimed features, but that, too, is irrelevant. Partly because of its many deviations from standard Pascal, but mainly because of its unreliability, I rate JRT Pascal as Not Acceptable. It is a product that could give "cheap" a bad name. After trying to use it, it is clear to me why its makers decided to lower its price. In my opinion they didn't lower it far enough.

DDJ

Clinic *(Continued from page 9)*

a barely-controlled-impatience heuristic to it. When at one level it has tried every possible encoding for a letter and failed, and when all those failures collided with the same previously-hashed word, and when as a result of those trials no forward progress has been made over previous trial encodings, the program declares the blocking word to be a "problem." It forces the algorithm to backtrack to the nearest point at which one of the letters of that word was encoded. This produced a significant speed-up in the cases where the program was already finding a solution (it then found *different* solutions), but didn't seem to help on those where it wasn't already working.

One set of keywords that the program did well on was a list of thirty-seven Pascal reserved words. It turned up a good encoding in just under a minute (Pascal/Z, 4MHz Z80 system). One encoding is shown in Table 1.

The whole subject of perfect hashing is one of those corners of systems programming that has the fascination of magic squares and other puzzles of recreational mathematics. You may enjoy analyzing the logic of our imperfect version of this particular algorithm. You can probably improve on it, or find a completely new approach. ▶▶J

(Listing begins on page 56)

Hash 'n Pascal

(Text begins on page 9)

Listing I

```
program perfect;
{-------------------------------------------------------------}
{ This program reads a set of words from a file and attempts  }
{ to find a perfect-hash encoding for them.  The logic follows}
{ that of Cook and Oldehoeft (SigPlan Notices, 9/82), but not }
{ very closely. If a suitable hash-encoding is found, a report}
{ is written to a file. This version compiles under PASCAL/Z  }
{ and runs under CP/M 2.2. Compiler dependencies are flagged  }
{ in comments.  Publication listing formatted by a modified   }
{ version of a pretty-printer by P. Grogono, PZUG disk #17.   }
{-------------------------------------------------------------}
const
    { the algorithm shouldn't be applied to a set of more than
 45 or so words, but let's be optimistic..}
    maxwords = 90;
    { the hash table size is 110% of the count of words,
 indexed by letter-codes from 0..size-1. }
    maxcode = 99;
    { length of the words in the set isn't very significant}
    maxchars = 16;

type
    wordspan = 1..maxwords; { indices over table of words }
    wordcnt = 0..maxwords;  { counts of words }
    hashspan = 0..maxcode;  { (max) range of valid hashcodes }
    codes   = hashspan;     { letter codes are hashcodes }
    letters = 0..25;     { uppercase ordinals less ord(A) }
    charspan = 1..maxchars; { indices over letters of a word }
    charcnt = 0..maxchars;  { counts of letters in a word }

    wordrec =
       record       { word info as held in the table }
          key : record    { key for initial table order }
            LA,      { initial/most-frequent letter }
            LZ : letters; { final/least-frequent letter }
            WL : charcnt  { word's length }
          end;
          hashval : codes; { tentative hash value }
          spelling : array[charspan] of char
       end;

var
    wordtab : array[wordspan] of wordrec;
    numwords : wordcnt; { actual count of words from file }

    hashtab : array[hashspan] of wordcnt;
    hashmax : hashspan; { actual size of table, 110% numwords }

    census : array[letters] of integer;{ frequency of use }
    double : array[letters] of boolean;{ eg., E as in ELSE }

    encoding : array[letters] of codes;
    encoded : array[letters] of boolean; { code assigned? }
    usedcode : array[codes] of boolean; { letter assigned? }

    problem : boolean; { global flag for fast backtracking }
    problt1,
    problt2 : letters; { letters causing intragroup conflict }
    blockwrd : wordcnt; { word that blocked a hash try }
    actdepth,  { current recursion level }
       depth : integer;  { deepest recursion level so far }

    OkSoFar : boolean; { status of main procedure's work }

procedure Initialize;

    var H : hashspan;
      L : letters;
      C : codes;
    begin
      for H:=0 to maxcode do
        hashtab[H] := 0;
      for L:=0 to 25 do
        begin
          census[L] := 0;
          double[L] := false;
          encoding[L] := 0;
          encoded[L] := false
        end;
      for C:=0 to maxcode do
        usedcode[C] := false;
      numwords := 0;
      hashmax := 0;
      problem := false;
      blockwrd := 0;
      actdepth := 0;
      depth := 0
    end;

procedure ReadAll;

{ Read all the keywords from the CP/M text file KEYWORDS into
    wordtab.  Order them by their keys, mainly so that we can
    detect conflicts -- words that have identical initials,
    finals, and lengths, like ORD and ODD.
  The format of the input file is : a word per line, uppercase,
    left-justified, no embedded blanks, no blank lines, no
    empty lines, no words of one letter or more than maxchars.
    NONE OF THESE REQUIREMENTS ARE CHECKED! }

var F : text;
    W, X, Y : wordcnt;
    stop : boolean;
    tw : wordrec;

    procedure ReadOne(var R:wordrec);
    { Read one word from the file and install it. Also keep
      the census of letter-frequency and double-ended words }
    begin
      readln(F,R.spelling);
      R.key.WL := maxchars;
      while R.spelling[R.key.WL]=' ' do
        R.key.WL := pred(R.key.WL);
      R.key.LA := ord(R.spelling[1])-ord('A');
      R.key.LZ := ord(R.spelling[R.key.WL])-ord('A');
      Census[R.key.LA] := succ(Census[R.key.LA]);
      Census[R.key.LZ] := succ(Census[R.key.LZ]);
      if R.key.LA=R.key.LZ then
        double[R.key.LA] := true
    end;

begin {ReadAll}
    reset('KEYWORDS',F) ; {PASCAL/Z dependent}
    W := 0;
    OkSoFar := true;
    while (not eof(F)) and (W<maxwords) do
      begin
        W := succ(W);
        ReadOne(wordtab[W]);
        stop := false;
        X := W;
        repeat { insert latest word into position by key }
          Y := pred(X);
          if Y>0 then
            if wordtab[X].key < wordtab[Y].key then
              begin
                tw := wordtab[Y];
                wordtab[Y] := wordtab[X];
                wordtab[X] := tw;
                X := Y
              end
            else stop := true { in position }
          else stop := true { at head of table }
        until stop;
        if (Y>0) then
          if wordtab[X].key=wordtab[Y].key then
            begin
              OkSoFar := false;
              writeln('Word ',
                      wordtab[X].spelling,
                      ' conflicts with ',
                      wordtab[Y].spelling)
            end
          else { not a conflict }
        else { at head of table, no conflict }
      end {while};
    if (W>0) then

      begin
        numwords := W;
        hashmax := (W + round(W/10.0))-1
{ n.b. "W+round(W/10.0)" is the number of slots in the table;
       that less 1 is the maximum index over the table and also
       the modulus used in the hash function.  The largest code
       that can be assigned is hashmax-1, because larger ones
       would just wrap around with the modulo function. }
      end
    else begin
        OkSoFar := false;
        writeln('file KEYWORDS not found or empty.')
      end
end{ReadAll};

procedure ReOrder;

{ Put the wordtab in the order described by Cook and Oldehoeft:
    (1) compute the frequency of the end-letters seen -- this
        has been done in procedure ReadOne/ReadAll.
    (2) assign a large value to any letter that appears at both
        ends of single word, as E does in ELSE.
    (3) for each word, if letter 2 (LZ) precedes letter 1 (LA)
        in the ordering, interchange the letters.
    (4) using letter 2 as the key, sort the word table into
        descending order by letter frequency.
    The result of this order is that letters such as E in ELSE
    (which have less choice of hash positions) are encoded
    first.  Next encoded are the most frequent letters (which
    are more likely to raise conflicts).

    Cook's and Oldehoeft's claim that the ordering also ensures
    that "letter 1 has been previously assigned a value" is
    clearly not true in every case (although it often will be);
    it is easy to imagine a set of keywords such that the end-
    letters of one keyword are disjoint from the end-letters of
    all other keywords. }

var L : letters;
    W, W1, W2 : wordspan;
    TL : letters;
    TW : wordrec;

begin
    for L := 0 to 25 do
      if double[L] then
        census[L] := census[L]+(numwords*2);
```

(Continued on page 58)

Hash 'n Pascal

(Listing continued, text begins on page 9)

```
    for W := 1 to numwords do
      with wordtab[w].key do
        if census[LA]<census[LZ] then
          begin
            TL := LA;
            LA := LZ;
            LZ := TL
          end;

    { sort the table -- no terribly quick sort is needed here
 as the backtracking code will easily dominate execution
 time regardless. }

    for W1 := 1 to numwords-1 do
      for W2 := numwords downto W1+1 do
        if census[wordtab[W1].key.LZ] <
           census[wordtab[W2].key.LZ] then
          begin
            TW := wordtab[W1];
            wordtab[W1] := wordtab[W2];
            wordtab[W2] := TW
          end;

end;

procedure AssignCode(var C : codes; L : letters);
{ Assign the first free code from C, to letter L }
begin
    while usedcode[C] do
      C := succ(C);
    encoding[L] := C;
    encoded[L] := true;
    usedcode[C] := true;
    write( chr(8), chr(C+ord('A')) )
end;

procedure FreeCode(C : codes; L : letters);
{ Back out of the assignment of code C to letter L }
begin
    encoded[L] := false;
    usedcode[C] := false;
    write(chr(8),'.')
end;
```

```
procedure TestProb(w1,w2 : wordspan);

{ Check the group of words wordtab[w1..w2] for an internal
  conflict. An intra-group conflict exists when, for some
  pair of words, (encoding[LA1]+WL1) is identical to
  (encoding[LA2]+WL2).  Since both words have the same LZ,
  no encoding of LZ will work.  If a conflict exists, note
  it in global "problem" and note LA1, LA2 in globals
  probltr1/2.  The main search routine will do a fast back-
  track to the point at which one of those was encoded. }

var
    conflict : array[codes] of wordcnt;
    cde : codes;
    wrd : wordspan;


begin
    problem := false;
    for cde := 0 to maxcode do
      conflict[cde] := 0;
    wrd := w1;
    while (not problem) and (wrd<=w2) do
      begin
        cde := encoding[wordtab[wrd].key.LA] +
               wordtab[wrd].key.WL;
        if conflict[cde] = 0 then
          begin
            conflict[cde] := wrd;
            wrd := succ(wrd)
          end
        else
          problem := true
      end;
    if problem then
      begin
        probltr1 := wordtab[wrd].key.LA;
        probltr2 := wordtab[conflict[cde]].key.LA
      end
end;

function TestGrp(w1, w2 : wordspan; c : codes): boolean;

{ test the success of the code c for the second letter of
  the group of words wordtab[w1..w2]. }

var
    hash : hashspan;
    success : boolean;
```

```
begin
    with wordtab[w1].key do
      hash := (encoding[LA] + WL + c)mod hashmax;
    if hashtab[hash]=0 then
      begin
        hashtab[hash] := w1;
        wordtab[w1].hashval := hash;
        if w1<w2 then
          begin
            success := TestGrp(succ(w1),w2,c);
            if not success then
              hashtab[hash] := 0
          end
        else
          success := true
      end
    else
      begin
        blockwrd := hashtab[hash];
        success := false;
      end;


    TestGrp := success
end;

function Letter1(w : wordspan) : boolean;
FORWARD;

function TryHash(w : wordspan) : boolean;

{ A group of one or more words, wordtab[w..?], have the same
  second letter.  The first letters of that group have all
  been given codes.  Try to find an encoding of their common
  second letters that will yield a perfect hash. }

var
    ltr : letters;  { the common second letter }
    cde : codes;    { the current code for that letter }
    wrd,     { the first word of the group }
    grp : wordspan; { ..and the last }
    success : boolean;
    blocksme : wordcnt;
    solidblock : boolean;
    olddepth : integer;

begin
    write('.'); { document recursion }
    actdepth := actdepth+1;
    if actdepth>depth then
      depth := actdepth;
    olddepth := depth;
    ltr := wordtab[w].key.LZ;

    { identify the span of the group -- the range of adjacent
      words with the same second letter. }

    grp := w;
    while (grp<numwords)
          and (ltr=wordtab[succ(grp)].key.LZ) do
      grp := succ(grp);

    { find out if there are any intra-group conflicts, and
      if so, don't bother trying to encode the second letter }

    if 0>(grp-w) then
      TestProb(w,grp);

    if problem then
      success := false
    else if encoded[ltr] then
      begin
 { the second letter of our group already has an
   encoding -- we don't have anything to say about
   it.  Try the effect of that encoding and, if it
   works at this level, recurse to the next group.
   If we fail at a lower level, that second-letter
   encoding has to be changed, so back out to that
   level directly. }
        cde := encoding[ltr];
        success := TestGrp(w,grp,cde);
        if success then { yippee! }
          if grp=numwords then
            writeln('!')
          else { more groups to test }
            begin
              if encoded[wordtab[succ(grp)].key.LA] then
                success := TryHash(succ(grp))
              else
                success := Letter1(succ(grp));
              if not success then
                for wrd:=w to grp do
                  hashtab[wordtab[wrd].hashval] := 0
            end
      end { of second-letter-has-encoding case }
    else
      begin
 { Our group's second letter hasn't received a code.
   We know that our group of words can't conflict with
   each other. Now try encodings of the common letter-2,
   looking for one that doesn't conflict with previously
   hashed words.  When we find one, we recurse to try
   the next group of words -- if any.}
        cde := 0;
        blocksme := 0;
```

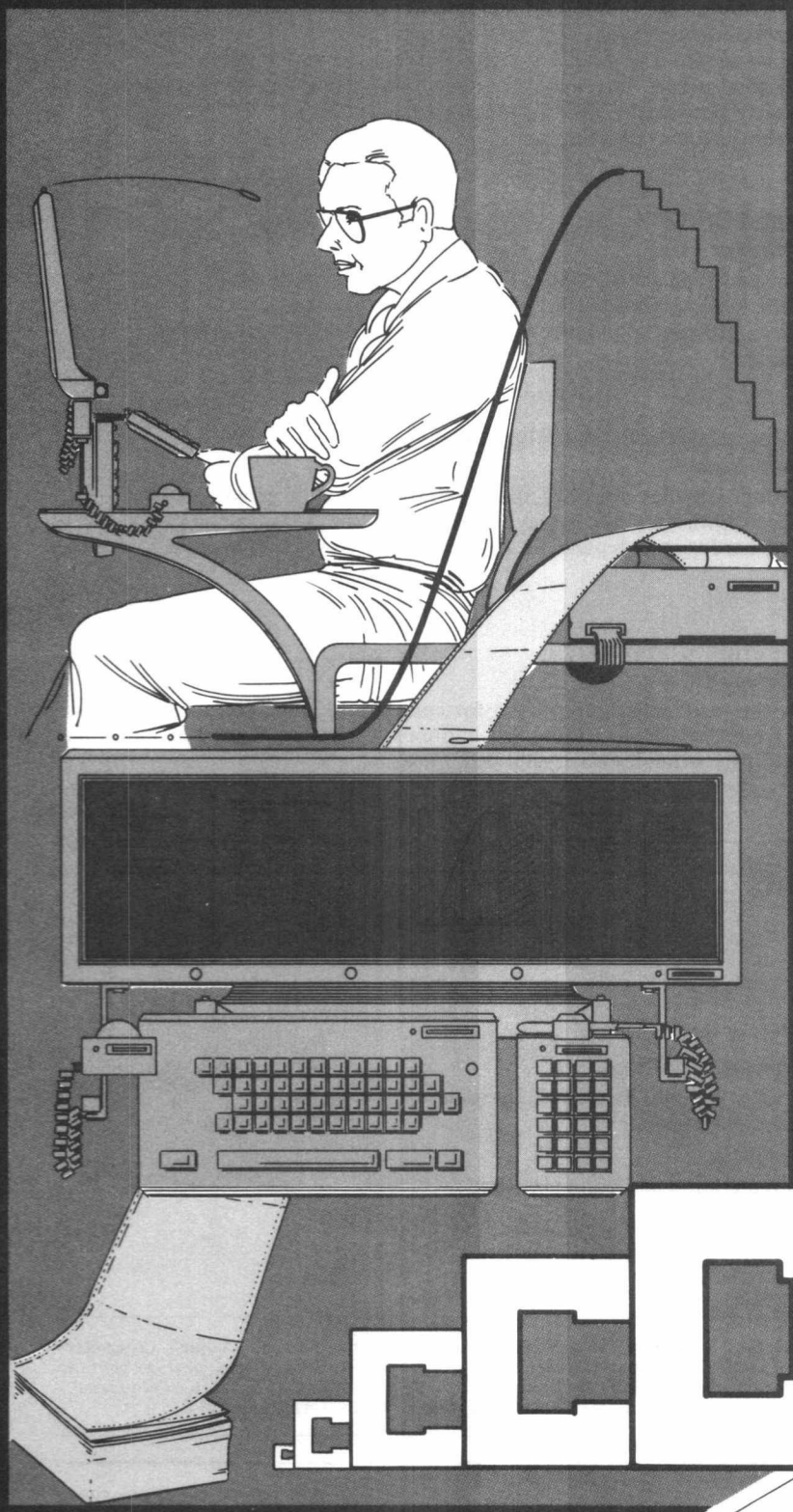
```
        solidblock := true;
        repeat

          repeat { find a code that works for this group }
            AssignCode(cde,ltr);
            success := TestGrp(w,grp,cde);
            if not success then
              begin
                FreeCode(cde,ltr);
                cde := succ(cde)
              end
          until success or (cde>=hashmax);

          if success then
            if grp=numwords then { yippee! }
              writeln('!')
            else { more words to hash }
              begin
                if encoded[wordtab[succ(grp)].key.LA] then
                  success := TryHash(succ(grp))
                else
                  success := Letter1(succ(grp));
                if not success then
                  begin { back out of group encoding}
                    FreeCode(cde,ltr);
                    cde := succ(cde);

                    for wrd:=w to grp do
                      hashtab[wordtab[wrd].hashval] := 0;
                    if blocksme=0 then
                      blocksme := blockwrd
                    else if blockwrd<>blocksme then
                      solidblock := false
                  end;
                if problem then
        { A lower group has an intragroup conflict.
          If we caused it, turn the flag off and
          try another code.  Else back out now.}
                  if (ltr=probltr1) or (ltr=probltr2) then
                    problem := false
              end
            else { no code works for our group }

          until success      {1 perfect hash complete }
                or problem   {2 must back out to higher level }
                or (cde>=hashmax);{3 no code worked for us }
          if (not success) and (not problem)
            and solidblock and (blocksme>0)
            and (depth=olddepth) {ie no progress} then
            begin
              problem := true;
              probltr1 := wordtab[blocksme].key.LA;
              probltr2 := wordtab[blocksme].key.LZ
            end;
        end;
      write(chr(8));
      actdepth := actdepth-1;
      TryHash := success
end;

function Letter1; { forward-declared above }

{ The first letter of wordtab[w] has not been encoded.
  Try to find a code for it, using TryHash to check out
  the effect on the current group of words.  The logic is
  much like that of TryHash, but simpler. }

var
    ltr : letters;
    cde : codes;
    success : boolean;

begin
    write('.'); { document recursion }
    ltr := wordtab[w].key.LA;
    cde := 0;
    repeat
      AssignCode(cde,ltr);
      success := TryHash(w);
      if not success then
        begin

          FreeCode(cde,ltr);
          cde := succ(cde)
        end;
      if problem then
        if (ltr=probltr1) or (ltr=probltr2) then
          problem := false
    until success      {1 perfect hash found below us}
          or problem   {2 have to back out to higher level}
          or (cde>=hashmax); {3 no code works for our letter }

    write(chr(8));
    Letter1 := success


end;

procedure ReportResults;
{ Write an ASCII file documenting the perfect hash we found.
    Since the program might have been left to run overnight,
    prompt the user to remount the diskettes and wait for
    somebody to notice us before writing. }

var f : text;
    L : letters;
    H : hashspan;
    W : wordcnt;
    X : char;


begin
    writeln('Press return when I can write a file...');
    readln(X);

    rewrite('ENCODING',f); {PASCAL/Z DEPENDENT}
    writeln(f,'Letter encodings:');
    for L := 0 to 25 do
      if encoded[L] then
        writeln(f,chr(L+ord('A')),
        ' = ',
        encoding[L]: 2 );
    writeln(f,'Hash values:');
    for H := 0 to hashmax do
      begin
        write(f,H:2,' = ');
        W := hashtab[H];
        if W>0 then
          write(f,wordtab[w].spelling);
        writeln(f)
      end
end;

begin { main procedure }
    Initialize;
    ReadAll;
    if OkSoFar then
      begin

        ReOrder;
        OkSoFar := Letter1(1);
        writeln;
        if not OkSoFar then
          writeln('Six years of computer time wasted...')
      end;
    if OkSoFar then
      ReportResults
end.
```

End Listing

#74 December 1982 $2.50 ($3.00 in Canada)

Dr. Dobb's Journal

For Users of Small Computer Systems



Small-C Compiler v.2

Interrupts and CP/M

for IBM's PC:

Custom Characters Lo-Res Color Graphics

Vector Generation Algorithm

Plus:
Small-C I/O Redirection and Command-Line Arguments

For Users of Small Computer Systems

December 1982 Volume 7, Issue 12

CONTENTS

Editor – Marlin Ouverson
Assistant Editor – Fritzi Lareau
Contributing Editors –
Dave Caulkins, Dave Cortesi,
Ray Duncan, Hank Harrison,
Gene Head, Michael Wiesenberg
Marketing Director – Craig Harper
Marketing Assistant – Beatrice Blatteis
Advertising Director – Carl Landau
Circulation Manager – Gloria Romanoff
Circulation Assistants –
Terri Marty, Linda Marohn
Art Director – Clifford West
Production Manager – Barbara Ruzgerian
Production Assistant – Jane A. McKean
Typesetter – Renny Wiggins
Cover Illustration – Al McCahon

© 1982 by People's Computer Company unless otherwise noted on specific articles. All rights reserved.

Subscription Rates: $25 per year within the United States; $44 for first class to Canada and Mexico; $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Writer's Guidelines: All items should be typed, double-spaced on white paper. Listings should be produced by the computer, using a *fresh, dark ribbon* on continuous white paper. Please avoid printing on perforations. Payment is in contributor's copies. Requests to review galleys must accompany the manuscript when it is first submitted. Authors may receive a copy of the complete writer's guidelines by sending a self-addressed, stamped envelope.

Donating Subscribers **Contributing Subscriber:** $50/year ($25 tax deductible). **Retaining Subscriber:** $75/year ($50 tax deductible). **Sustaining Subscriber:** $100/year ($75 tax deductible). **Lifetime Subscriber:** $1000 ($800 tax deductible). **Corporate Subscriber:** $500/year ($400 tax deductible, receives *five* one-year subscriptions).

Contributing Subscribers: Thomas H. Grohne, G. V. Elkins, Robert M. Connors, William J. McEown, Robert C. Luckey, Robert N. Stabler, DeWitt S. Brown, John B. Palmer, Burks A. Smith, Transdata Corporation, Mark Ketter, John W. Campbell, Friden Mailing Equipment, Howard E. Decker, Jim Wilson, Frank Lawyer, Chris Pettus, Paul Lutus, Ben Goldfarb, Rodney Black, Steven Weiss, Pat Phelan, John Brodie, James Fleming, Unison World, Thomas Davis, Jan Steinman, Ronald E. Johnson, G. Hunter, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor, Jr., Ted A. Reuss III. **Sustaining Subscribers:** Pete Roberts, Steven Fisher Enterprizes. **Lifetime Subscriber:** Michael S. Zick.

Foreign Distributors **UK & Europe:** Hofacker-Verlag, Tegernseer Strass 18, D-8150 Holzkirchen, West Germany. **Asia & Australia:** ASCII Publishing Co. Ltd, 4F Segawa Building 5-2-2, Jingumae, Shibuya-ku, Tokyo 150, Japan.

Articles

16 Small-C Compiler, v.2
by J. E. Hendrix
The C programming language is still gaining momentum as a powerful alternative to assembly language. The public domain Small-C compiler published in *DDJ* was clearly a good subset, but was still a bare-bones compiler. This article introduces the greatly improved Small-C version two. The last half of the program listing will be published next month.

54 Interrupts and CP/M
by Alan Bomberger
The problems of "type ahead" under CP/M have kept more than one programmer awake at night looking for a better solution. Author Bomberger illustrates the use of interrupts to provide fully-buffered console input which can be used by CP/M when needed.

58 A Simple Vector-Generation Algorithm
by Marek W. Michalski
Anyone writing graphics routines must confront vector generation head-on. This paper deals with graphics displays for eight-bit microprocessors. He has tried to find a fast, hardware-independent algorithm and provides an illustration of its implementation.

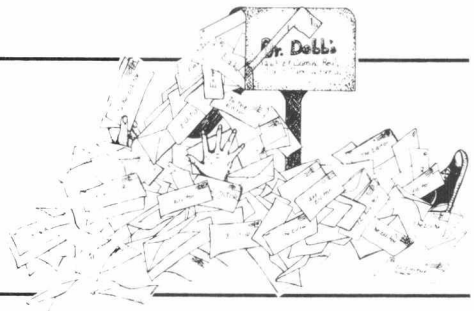
60 Fifth-Generation Computers
by Richard Grigonis
What will be the next evolutionary step for computers? What should it be? This contributor from the forward-looking Children's Television Workshop takes an educated look into technology's crystal ball. . .

Departments

9 Letters
10 Editorial
Two Years Before the Masthead
11 16-Bit Software Toolbox
Custom character sets and lo-res color graphics for IBM's PC; hex and binary conversion.
62 Dr. Dobb's Clinic
Small-C I/O redirection and command-line arguments.
66 Of Interest

Dr. Dobb's Journal (USPS 307690) is published twelve times per year by People's Computer Company, Box E, 1263 El Camino Real, Menlo Park, CA 94025. Second class postage paid at Menlo Park, California 94025 and additional entry points. Address correction requested. Postmaster: send form 3579 to Box E, Menlo Park, California 94025 · 415/323-3111

LETTERS



File Transfer Bugs

Dear Dr. Dobbs:

I recently acquired a new 8" disk system to replace the 5" drives I have been using, and after a friend helped me transfer all my programs from the old disks to the new ones using a modem program, one of us said, "What we need is a program that transfers the whole disk automatically, instead of one program at a time." The next day, when the August *DDJ* came, there it was – the SEND/RECEIVE file transfer utilities by Lawrence Hughes.

Since I have been using the MFT program of Mr. Hughes for some time, and I am sure that the SEND/RECEIVE utilities will be useful in the future, I thought that now would be a good time to write a note of appreciation for these programs being made available. Thanks a lot to both Mr. Hughes and *DDJ*.

I think there are a couple of errors in the listing for the SEND program, however. At the end of the routine "send:" just before "send0:" there should be a "jmp send8" statement. Without this statement, the program tells you that no files were found, and then tries to send them anyway instead of aborting.

The other error appears in the "send7" routine where the following code appears:

```
decr m
jmp  send0
jmp  sendx
```

Obviously the second jump would never be executed. I believe that the first jump should be:

```
jnz  send0
```

As written, it appears to me that the program would continue to try to send files forever.

Perhaps Mr. Hughes could comment on the correctness of these observations, as I have not really tested the program yet.

Sincerely,
Robert Wilcox
920 N. Washington St.
Owosso, MI 48867

Issuing Forth

People:

My first issue of *DDJ* just came yesterday. I was delighted to find that you had a feature on Forth (issue #71).

You have quite an impressive magazine – I should have gotten a subscription long ago.

Regards,
Dr. Alan D. Wilcox
12211 Westwood Hills Dr.
Herndon, VA 22071

Sundries

Marlin Ouverson,

I'm in agreement with the public domain principle. I think human progress is too important to be hampered by the inefficiency of too much private (secret) enterprise. Luckily, the vast quantity of software should reduce its own price rapidly, just by virtue of there being so much. Though my agreement is largely political-style, I am also currently interested in electronics and computers.

The problem for me is that computers are very complicated, as I discovered when at a store called Puter-Parts here in Seattle. The owner took a lot of time to help me program an EPROM I bought there at a reasonable price. I was really happy about his concept of thoroughly aiding even small customers, but embarrassed by my inability to understand Z80 machine language even by studying quite hard. There's a krillion things you have to know! I wish him, and others who bring their knowledge into the public domain, success.

Politically, I see the "vision" that there are gigantic benefits for all in the world when technology develops answers to social problems – the way technology can achieve those answers is by people knowing it and inventing the solutions.

Thanks again for maintaining the humanitarian public-domain principle; when my ZX81 comes (if it ever does!) I'll write you and tell the tricks and difficulties others might find useful.

Sincerely,
Jack Scott
3 Enatai Drive
Bellevue, WA 98004

About JRT Pascal

10-15-82

Editor:

You asked me to reply to the very critical and insulting Software Review of JRT Pascal by D. Cortesi in the November *DDJ*.

The hundreds of complimentary letters and Thank You notes I have received and the outstanding reviews published in *Infoworld*, *Interface Age*, and *Creative Computing* reflect the opinion of the overwhelming majority of JRT Pascal users.

JRT Systems will do no further business with the People's Computer Company.

James R. Tyson
JRT Systems, Inc.
P.O. Box 22365
San Francisco, CA 94122

P.S.: Please do not edit this reply.

Editor's note: The following are the only two letters we have received (as of our deadline for this issue) which mention JRT Pascal. Both were received after our review of that product was completed but before the issue containing it was mailed; hence they did not affect our review, nor vice versa.

Dr. Dobb's Clinic

Dear Mr. Cortesi:

Since you are scraping the bottom of your mailbag, I would like to contribute the following complaint.

The Pascal system that JRT Systems have been heavily promoting for the past several months for $29.95 is highly overpriced. By how much? Exactly $29.95. Why? JRT Pascal is unable to correctly multiply floating-point numbers in the form X.Y * 0.0, giving answers of X.Y instead of 0.0, as the enclosed listing reveals.

Sincerely,
Andrew Omori
4009 Chandler Blvd.
Burbank, CA 91505

(See listing page 70)

Dear Dr. Dobb's Clinic:

I have an array of problems, discoveries and hypotheses that I would like to air and see what returns. I should begin by saying that my system is a Z-80 CP/M system. It's a single-board system with a Scull-Tek circuit board from Romac Computer Equipment of Naperville, Illinois. My problems revolve around two software products: JRT Pascal from JRT Systems (advertised in *Dr. Dobb's*) and DDT from Digital Research. I should hasten to say that I am not convinced these products are at fault for my problems.

(Continued on page 70)

Two Years Before the Masthead

Nearly three years ago, Ron Cain used these pages to place a Small-C compiler in the public domain. It was many people's introduction to C as such a reasonable alternative to assembly language. Since then we have published many letters and articles detailing bug fixes, patches, and extensions – but mostly praise – for the compiler.

DDJ is pleased to present in this issue a new version of the Small-C compiler. J. E. Hendrix developed it with the support of Ron Cain and a number of other *DDJ* subscribers. Code written under version one should be compilable by this one; both are upwardly compatible with full C. By perusing the listing (which will be continued next month) and its accompanying article, many of you will note how Small-C has grown – but it is faster, too, and the programs it produces typically are smaller than previously.

* * *

This is the last issue for which I will serve as *Dr. Dobb's* editor. Any regret at my decision to leave this post is tempered by pleasant recollections of my past two years in association with the best minds and friendliest people in microcomputing. I was hired half-willingly for an editorial position which I imagined would be as sterile as the silicon laboratories not far from our offices. What I found was just the opposite, a loose-knit clan of idealists, hackers, and professionals, brought together by their mutual drive to see powerful computers and software tools available to the average person. It has been a greater pleasure to work with them than I could have hoped for.

Dr. Dobb's Journal is very different from any other microcomputer publication, because of the extent of its commitment to the readers. *DDJ* has never been content to rest on the laurels of its considerable reputation, or to fill its pages with middle-of-the-road articles. Nor do we attempt to tell our readers what is important. We only provide space where people and facts can speak for themselves. That sounds simple enough, but it became apparent very early during my stay here that the job would not be an easy one.

Doc's editor faces an enormous task, only part of which consists of filing through a bushel of press releases each month, digging elsewhere for as-yet-unannounced items, proofreading and more proofreading, all while seeking good programmers willing to put the fruit of their labors into the public domain. Fortunately, the *DDJ* readership itself is the best source of technical support and microcomputer expertise any editor could ask for. It is that, and not the sole efforts of any single individual, which provides the continuously high-quality contents.

* * *

Some exciting items already are planned for the next several issues. *DDJ* will be publishing Augusta, an Ada subset, in four installments. Ada (the name is a trademark of the U.S. Department of Defense) is going to be making some big waves, but the DoD has resisted approving any subsets of the language. The size of the full implementation causes most micros to shrivel with anxiety, so this will be your opportunity to get a head start at learning the language. Programs written in Augusta will be needed in *DDJ*, so don't waste any time getting it up and running on your machines.

The far-seeing people at Project Xanadu have some material lined up for Doc Dobb's, too. In any dry season, the sparks of their creativity are sure to set brushfires in all directions. Those folks are bent on proving that no matter how big the dream, it can be coded and installed on a computer. Keep an eye on their work – it's fascinating.

The *DDJ* colony of contributing columnists is lining up even more useful routines, bug fixes, patches and puzzlements for the coming year. The 16-bit Software Toolbox will continue providing useful routines for the 8086/88 and will add material for other processors, especially the 68000 (and, if enough readers clamor for it, the Z-8000). The CP/M Exchange will still be promoting telecommunications and the RCP/M approach to downloading software, but will branch out to answer general questions and to provide tips about CP/M. The Clinic will keep titillating readers with its eclectic assortment of algorithms and other oddments.

* * *

We who have had some degree of involvement with *DDJ* and People's Computer Company (PCC) modestly think of this publication as the lever which, with the slightest degree of pressure, just might move the world a bit. By serving the high end of the technological spectrum, some of our efforts do find their way by mysterious means into products and services which help people. This happens when one of our readers who sees the true potential of computers, and some piece of software we publish, puts them together in new ways.

Dr. Dobb's Journal is primed and ready for what the future brings. Microcomputers' potential for humanitarian use is just beginning to be tapped in educational and recreational settings, health care facilities, work environments and the home. More important, the old view of computers as untouchable pawns of the world's power merchants has been replaced by the knowledge that they are only machines, tools as accessible as any other, designed to serve people and, hopefully, to improve the quality of everyday life.

Happy computing.

– Marlin Ouverson

16-BIT SOFTWARE TOOLBOX

by Ray Duncan

Concerning Column Topics

Several readers have already written to ask why "excessive" attention is being devoted to the Intel 8086/8088/8087 microprocessor family. The main reason may be seen in Table 1, i.e., the Intel line has an overwhelming predominance of the 16-bit user base. The secondary reason, not unrelated to the first, is that until now most Z8000 and 68000 systems sold have been expensive, integrated packages with proprietary software and not appropriate for the personal computing market. These drawbacks are going to disappear rapidly for the 68000 as Cromemco and Godbout begin to ship their new S-100 bus, 68000-based CPU boards and as Digital Research CP/M-68K becomes widely available. I personally feel that time has already passed by the Z8000, mainly due to poor marketing strategy on the part of Zilog, but I would be very interested to hear counterarguments.

Custom Character Sets for the IBM PC

Most owners of the IBM Personal Computer have heard rumors that its operating system permits installation of a user-defined character set for the video display. Like many of the other sophisticated features of the PC, this capability is alluded to here and there throughout the IBM Technical Manual but the procedure is not spelled out in detail. In this column, I will explain the exact steps to follow to create a new video character font. I hope that, in return, readers will use these techniques to implement some useful collections of symbols and contribute them back into the public domain via *DDJ*.

Software-defined character sets can only be displayed using the color/graphics interface board in "graphics" mode. This is because the ASCII character codes are translated by software into bit patterns written directly into the video memory map. In contrast, when using the monochrome video interface or the color/graphics display in "text mode," the ASCII codes themselves are stored into the video memory buffer and are converted by the hardware into a pixel array according to the contents of a character generator PROM. Consequently, there is no way while in text mode for the user to affect the shape of a given character except by burning a new PROM and soldering it onto the interface board (not an approach that I would recommend).

When the color board is in graphics mode, and the video driver is called with an ASCII character code to be displayed, it uses that code as an index into a bit table that is stored in memory. A text character is displayed as an 8 by 8 matrix, so each ASCII code corresponds to eight bytes in the table. The eight bits of each byte are equivalent to a horizontal line of eight pixels, the most significant bit corresponds to the leftmost pixel of a group and the least significant bit to the rightmost pixel. The byte with the lowest memory address corresponds to the top scan line of a character, and so forth to the eighth byte of a set (with the highest address) which maps onto the bottom scan line of the character. Some fancy conversion occurs when text is displayed in color graphics mode, where each pixel is two bits, but that need not concern us here since it is handled automatically by the resident routines.

The bit table for the first 128 ASCII codes (00-7FH) is present as part of the ROM software of the PC, starting at address F000:FA6EH (FFA6EH). The display patterns for these characters cannot be modified. However, IBM has left the upper 128 ASCII codes (80-FFH) vacant and the user can define these by creating a supplementary bit table. A dedicated pointer location in lower memory has also been provided so that you can tell the operating system's video driver where to find your special bit map. Then the application software can switch back and forth between IBM's character set and a custom character set simply by toggling the most significant bit of the ASCII mode.

The general steps that must be followed to install a new character set are:

1. Sit down with pencil, graph paper, and a bit eraser, and diagram out each of your desired characters in an eight by eight matrix (see Figure 1). Note that the IBM character set uses the bottom row and rightmost two columns of pixels as spacing; it's a good idea to follow this convention so that the display will appear uniform.

2. Convert each of the character grids into the equivalent bit pattern, and thence into eight bytes of data (see Figure 2).

3. Assign each character to an ASCII code between 128 (80H) and 255 (0FFH), then insert the eight bytes for that character into a table. Calculate the table starting address for a given character code as follows:

Table offset = (ASCII code - 128)*8

4. Using DEBUG, enter the completed table (1024 bytes) into RAM, then write it into a disk file for safekeeping.

5. So that it can be used by an application to generate a video display, get the bit table into RAM at a location that will not be destroyed by other programs (don't get exasperated, I'm going to give you a way to do this).

6. Place the offset address of your bit table in the location 0000:007CH (0007CH), and the segment address of the table into 0000:007EH (0007EH). This is referenced by the PC's video driver when it needs to display an ASCII code in the range 80-0FFH.

At first glance, step 5 above looks most difficult by far. However, reading the fine print in the DOS manual, we find that a facility is provided (using interrupt service 27H) to let a program terminate without freeing up its memory for use by subsequent programs. Thus, all we need is a little utility that will read a specified character generator bit table from a disk file into RAM, patch the dedicated vector location to point to the table, and notify the operating system as it exits that the memory containing the table must be reserved from use by other programs. The code for this utility, named "CLINK," is provided in Listing #1.

CLINK expects the user to supply the name of the character table file in the original command line, so that the default file control block is properly set up by the operating system before CLINK begins executing. First the program tries to open the character file, and exits with an error message if the file does not exist. Next the bit table is read into an internal buffer; the file is expected to contain the full 1024 bytes even if you are not using all of the character codes. Finally, the operating system link address is set to point to the table buffer, the program displays a success message, and then performs the special exit back to PC-DOS. The net amount of memory made unavailable for use by other programs is about 1400 bytes.

You will probably find it quickest to create the CLINK.COM file by entering

the object code directly with DEBUG, therefore I have outlined the necessary steps in Listing #2. Once CLINK is installed, loading a character table from disk into RAM and linking it to the operating system is quite simple. For example:

```
A>CLINK RUSSIAN.CHR
   character table loaded and linked.
A>
```

User Groups

Joseph Boykin responded almost instantaneously to the first appearance of this column by advising us of the formation of the Seattle Computer Products Users' Group, which is intended to promote sharing of knowledge and software between users of 8086/MS-DOS SCP microcomputers. The group intends to provide a dial-up bulletin board, diskettes of public domain software, and a periodic newsletter. Membership dues are stated to be $10.00 per year. For more information contact Frank Warren, 25190 Cypress Ave. #213, Hayward, CA 94544.

Other User Groups for 16-bit microcomputers are also invited to make their existence known by sending information to me c/o *DDJ*.

Hex and Binary Interconversion

Robert Blair of Walnut, California sent in a nice set of subroutines for Intel 8086/88 that will allow conversion of an 8-bit binary value into a hex ASCII string and vice versa (see Listing #3 and #4). This code will be very useful for display of data in debugger programs, disk file utilities, and the like. Such contributions really warm my heart. Please keep the listings coming!

IBM PC Low-Res Color Graphics

The following is reproduced verbatim from another windfall that arrived courtesy of John J. Seal of Franklin, Indiana.

"On page 2-52 of the IBM PC Technical Reference Manual is described a low-resolution 160 x 100 16-color graphics mode. This mode can be used to create graphics in which all 16 available colors are displayable simultaneously, as they are in text modes. Unfortunately, this mode is not supported by BASIC or by the character generators in the BIOS. The manual cryptically states that it 'requires a special memory map and set up to be defined later.'

"This is how it is done:

"1. The 160 x 100 'graphics' mode is actually a text mode! Set up the Color Graphics adaptor for 80 x 25 alpha color mode (not 40 x 25 as stated on page 2-59).

"2. Set all the Character Codes (CC) in the display buffer to 0DEH. This character is split vertically with 'background' on the left and 'foreground' on the right, thus giving 160 half characters across the width of the screen.

"3. Set the 6845 CRT Controller to only display one-fourth of each row, i.e. two scan lines, but to display four times as many rows. This gives 100 quarter-characters from top to bottom.

"4. The color of each pixel is controlled by the attribute bytes (AT) in the display buffer, two pixels per byte, just as in text mode. The high nibble specifies I,R,G,B for the background (left half) and the low nibble specifies I,R,G,B for the foreground (right half).

"From this we can see that 'special set up' means selecting a non-standard text mode and 'special memory map' means presetting all the character codes in the display buffer. The '8000 byte Read/Write memory' mentioned on page 2-52 consists of all the odd-numbered bytes from B8000H to BBFFFH."

Mr. Seal also included a BASIC program (see Listing #5) which demonstrates how to set up the 160 x 100 16-color graphics mode, and a function for computing offsets into a display buffer based on X and Y coordinates.

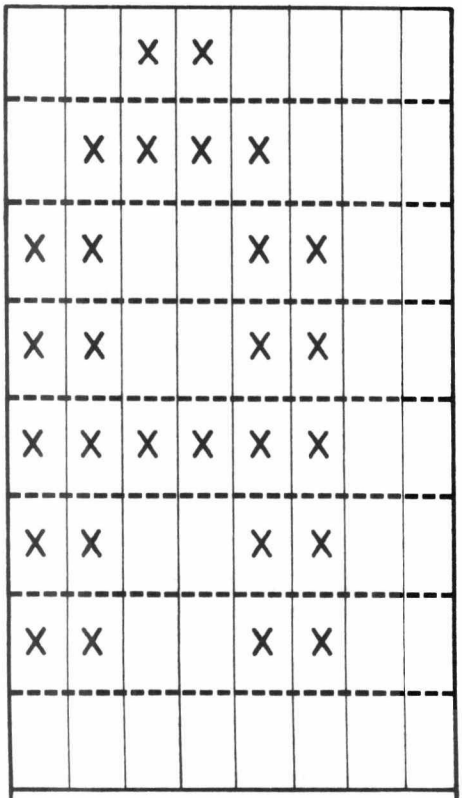


Figure 1.

The character "A" diagrammed as an eight by eight dot matrix (copied from the supplied IBM character set). Each of the symbols in a custom character set must be laid out in a similar fashion.

Bit pattern for character "A" from Figure 1:

```
0 0 1 1 0 0 0 0
0 1 1 1 1 0 0 0
1 1 0 0 1 1 0 0
1 1 0 0 1 1 0 0
1 1 1 1 1 1 0 0
1 1 0 0 1 1 0 0
1 1 0 0 1 1 0 0
0 0 0 0 0 0 0 0
```

which translates to the following eight data bytes which are inserted into the character table:

```
30 H
78 H
CC H
CC H
FC H
CC H
CC H
00 H
```

Figure 2.

Mapping the pattern shown in Figure 1 into a set of eight data bytes for the character table.

Microprocessor	Units	% of market
Intel 8086/88	718,000	78%
Motorola 68000	139,000	15%
Zilog Z-8000	61,000	7%

Table 1.
1981 Sales of various 16-bit microprocessors.

Highly Recommended Software

One of the most pleasant surprises of the year is "PC-TALK," a telecommunications program for the IBM Personal Computer by Andrew Fluegelman. PC-TALK is written in BASIC and is designed for use with the Hayes Smartmodem, though it can easily be used with other direct-connect or acoustic modems too. The program is slick, friendly, reliable, and completely self-documenting via well-designed menus and on-line help files. Support is provided for automatic dialing, customized function key definitions, Compu-Serve and Source special protocols, and uploading/downloading of files. PC-TALK is a quantum leap in quality over the IBM Asynchronous Communications Program. Distribution of the program is based on a novel concept called "Freeware," wherein the user has the right to freely copy the program and pass it on. Send $25 to The Headlands Press Inc., P.O. Box 862, Tiburon, CA 94920.

DDJ

(Listing begins on page 14)

CLINK (Text begins on page 11)

Listing One

```
                        title   'Load & Link Char. Table'
                        pagewidth       78
                        cseg
                        org     100h
                ;
                ;
                ; Load and Link Character Table
                ; (for IBM PC under PC-DOS)
                ; Version 1    September 20, 1982
                ;
                ; Ray Duncan, Laboratory Microsystems
                ; 4147 Beethoven Street, Los Angeles, CA 90066
                ;
                ;
005C            fcb     equ     005ch   ;file control block
000D            cr      equ     13      ;carriage return
000A            lf      equ     10      ;line feed
                ;
                ;
0100 8CC8       link:   mov     ax,cs   ;initialize segment
0102 8ED8               mov     ds,ax   ;registers
0104 8EC0               mov     es,ax
0106 BA5C00             mov     dx,offset fcb
                                        ;using default fcb,
0109 B40F               mov     ah,15   ;open char. table file
010B CD21               int     21h
010D 0AC0               or      al,al   ;check if successful
010F 740B       011C    jz      link1   ;jump,file opened ok
0111 BA4F01             mov     dx,offset linka
0114 B409               mov     ah,9    ;open failed, print
0116 CD21               int     21h     ;error message
0118 B400               mov     ah,0    ;then exit
011A CD21               int     21h     ;back to PC-DOS
                ;
011C BA8601     link1:  mov     dx,offset cbuf
011F B41A               mov     ah,26   ;set disk transfer
0121 CD21               int     21h     ; address
0123 BB5C00             mov     bx,offset fcb
0126 C7470E0004         mov     14[bx],1024 ;and rec. length
012B BA5C00             mov     dx,offset fcb
012E B414               mov     ah,20   ;read char. table
0130 CD21               int     21h
0132 BA6001             mov     dx,offset linkb
0135 B409               mov     ah,9    ;print success message
0137 CD21               int     21h
0139 BB7C00             mov     bx,07ch ;set OS vector to point
013C 33C0               xor     ax,ax   ;to new char. table
013E 8ED8               mov     ds,ax
0140 8CC8               mov     ax,cs   ; segment address
0142 894702             mov     2 [bx], ax
0145 B88601             mov     ax,offset cbuf
0148 8907               mov     [bx],ax ;offset address
                                        ;now point to last address
                                        ;of program and exit,
014A BA8605             mov     dx,offset cbuf+1024
014D CD27               int     27h     ;leaving table resident
                ;
014F 0D0A6E6F2073 linka  db     cr,lf,'no such file',cr,lf,'$'
     756368206669
     6C650D0A24
0160 0D0A63686172 linkb  db     cr,lf,'character table '
```

```
0002  B9 0004             MOV   CX,4
0005  8A F2               MOV   DH,DL             ;split byte
0007  D2 EA               SHR   DL,CL             ;shift for translate
0009  80 E6 0F            AND   DH,00001111B      ;remove high bits for translate
000C  2A FF               SUB   BH,BH
000E  8A DA               MOV   BL,DL
0010  2E: 8A 97 001F R    MOV   DL,CS:TRANSLATE_HEX [BX] ;translate
0015  8A DE               MOV   BL,DH
0017  2E: 8A B7 001F R    MOV   DH,CS:TRANSLATE_HEX [BX] ;translate
001C  5B                  POP   BX
001D  59                  POP   CX
001E  C3                  RET                     ;return to caller
001F  30 31 32 33 34 35   TRANSLATE_HEX DB '0123456789ABCDEF'     ;translate table
      36 37 38 39 41 42
      43 44 45 46
002F                      BINHEX  ENDP
```

HEXBIN

Listing Four

```
002F                    PAGE
                HEXBIN  PROC  NEAR
                ;
                ; Convert HEX to Binary
                ;
                ; Call with      DL = first HEX digit
                ;                DH = second HEX digit
                ;
                ; Returns with   DL = converted byte
                ;                CARRY FLAG = 0     valid input data
                ;                CARRY FLAG = 1     invalid input data
                ;                Other registers preserved.
                ;
002F  51                PUSH  CX
0030  53                PUSH  BX
0031  8A DA             MOV   BL,DL             ;get first byte to test
0033  E8 004D R         CALL  HEXBIN_TEST       ;check if valid HEX digit
0036  8A FB             MOV   BH,BL             ;save first digit
0038  8A DE             MOV   BL,DH             ;get second byte to test
003A  E8 004D R         CALL  HEXBIN_TEST       ;check if valid HEX digit
003D  B9 0004           MOV   CX,4
0040  D2 E7             SHL   BH,CL             ;shift bits (first digit)
0042  80 E3 0F          AND   BL,00001111B      ;clear extra bits (second digit)
0045  8A D7             MOV   DL,BH             ;move digit
0047  0A D3             OR    DL,BL             ;combine bits
0049  F8                CLC                     ;set good return
004A  5B                POP   BX
004B  59                POP   CX
004C  C3                RET                     ;return to caller
004D                    HEXBIN_TEST:
004D  80 FB 30          CMP   BL,'0'
0050  72 22             JB    HEXBIN_ERROR
0052  80 FB 39          CMP   BL,'9'
0055  77 01             JA    $+3
0057  C3                RET                     ;return if 0 thru 9
0058  80 FB 66          CMP   BL,'f'
005B  77 17             JA    HEXBIN_ERROR
005D  80 FB 61          CMP   BL,'a'
0060  72 04             JB    $+6
0062  80 C3 09          ADD   BL,9              ;adjust if a thru f
0065  C3                RET
```

```
      616374657220
      7461626C6520
 0172 6C6F61646564              db      'loaded and linked',cr,lf,'$'
      20616E64206C
      696E6B65640D
      0A24
                          ;
   0186                   cbuf    equ     $
                          ;
                                  end


END OF ASSEMBLY.  NUMBER OF ERRORS:    0.  USE FACTOR:  0%
```

CLINK.COM

Listing Two

```
A>DEBUG
-F 100 L 100 0                          (initialize everything to zero)
-E 100                                  (now enter object code...)
04DB:0100  00.8C   00.C8   00.8E  ...   (and so forth for entire program)
-
-N CLINK.COM                            (set up file control block)
-R BX                                   (set BX and CX to file length)
BX 0000
:100
-R CX
CX 0000
:100
-W                                      (write object file and exit)
Writing 0100 bytes
-Q

A>
```

BINHEX

Listing Three

```
                        PAGE
0000                BINHEX  PROC  NEAR
                    ; Utility subroutines for 8086/8088
                    ; Robert Blair
                    ; September 3, 1982
                    ;
                    ; Convert Binary to HEX
                    ;
                    ; Call with      DL = byte to convert
                    ;
                    ; Returns with   DL = first HEX digit
                    ;                DH = second HEX digit
                    ;                Other registers preserved.
                    ;
0000 51                         PUSH  CX
0001 53                         PUSH  BX
```

(Continued on top of page 14)

```
0066 80 FB 41                   CMP   BL,'A'
0069 72 09                      JB    HEXBIN_ERROR
006B 80 FB 46                   CMP   BL,'F'
006E 77 04                      JA    HEXBIN_ERROR
0070 80 C3 09                   ADD   BL,9              ;adjust if A thru F
0073 C3                         RET                     ;return to HEXBIN routine
0074                HEXBIN_ERROR:                       ;error return
0074 F8                         CLC
0075 F5                         CMC                     ;set carry to indicate error
0076 5B                         POP   BX              ;POP the call for this subroutine
0077 5B                         POP   BX              ;restore registers
0078 59                         POP   CX
0079 C3                         RET                     ;return to caller
007A                HEXBIN  ENDP
```

Lo-Res Setup

Listing Five

```
100 '
110 ' Set up low resolution 160 x 100 16-color graphics mode
120 '
130 SCREEN 0 : WIDTH 80                 ' set up 80 x 25 alpha
140 KEY OFF : CLS                       ' clear screen
150 OUT &H3D8,9                         ' disable blink attribute
160 A%=&H3D4 : D%=&H3D5                 ' crt controller ports
170 OUT A%,4 : OUT D%,&H7F              ' vertical total
180 OUT A%,6 : OUT D%,&H64              ' vertical displayed
190 OUT A%,7 : OUT D%,&H70              ' vertical sync position
200 OUT A%,9 : OUT D%,1                 ' max scan line address
210 DEF SEG=&HB800                      ' display buffer segment
220 FOR CC%=0 TO &H3FFE STEP 2          ' all character codes
230 POKE CC%,&HDE : NEXT                ' set to special character
240 FOR AT%=1 TO &H3FFF STEP 2          ' all attributes
250 POKE AT%,0 : NEXT                   ' preset to black
260 '
270 ' Compute offset into display buffer, based on X and Y
280 '
290 DEF FNOFFSET%(X%,Y%)=160 * Y% + (X% OR 1)
300 '
310 END
```

End Listing

Small-C Compiler, v.2

Ron Cain stirred up considerable interest when he demonstrated (*DDJ* #45) how effective even a small compiler can be. Many people were inspired to adapt his compiler to their systems. My own implementation resulted in Small-VM (*DDJ* #61) which interfaces the compiler, and programs compiled by it, to North Star DOS. The Small-Shell command processor (*DDJ* #63) further enhanced this environment, giving it even more of a UNIX flavor.

I immediately used the compiler to create a set of utility programs based on the ones presented in the book *Software Tools* by B. W. Kernighan and P. J. Plauger. The original Small-C compiler was more than adequate for the task and clearly superior to the RATFOR language used in the book. But it was a bare-bones compiler, literally begging for improvement. It seemed, too, that a little effort devoted to code optimizing might yield valuable benefits. So, with encouragement from Ron Cain and the Doctor, I set out to produce a second version. Listing #1 contains the results.

I am in debt to Ron Cain, who presented the original compiler for all to do with as they pleased and who provided much invaluable guidance for this project. In addition, Neal Block (P.O. Box 8829, Fountain Valley, CA 92708) was also very helpful by providing many specific suggestions relating to performance, generalizing the computer for use on machines of various word lengths, and approaches to implementing the new control structures. I hope before long to see him come forth with an assembler and linking loader written especially for Small-C. Finally, Dr. James Van Zandt (26 Shelton St., Nashua, NH 03062) suggested replacing the serial table searches with hash searches for greatly improved speed.

Differences

This version of the compiler differs from the original one in the following respects:

1. Code optimization has been added. Programs are now typically 15 to 30 percent smaller than before.

2. Data initialization is supported for global variables, arrays, and pointers. Uninitialized objects default to binary zero.

3. Constant expressions are now evaluated at compile time. They may also be used as array dimensions, data initializers, and in "case" statements.

4. The #ifdef, #ifndef, #else, and #endif statements are supported with nesting.

5. The "extern" storage class is supported for global variable, array pointer, and function declarations. The effect is to declare them as external references to the assembler.

6. A mechanism for passing an argument count to called functions is provided.

7. The following new statements are now supported:
   a. for
   b. do/while
   c. switch/case/default
   d. goto

by J. E. Hendrix

Copyright © 1982 by J. E. Hendrix, Rt. 1, Box 74-B-1, Oxford, MS 38655.

8. Lists of expressions are now premitted.

9. The assignment operators |=, ^=, &=, +=, -=, *=, /=, %=, >>=, and <<= are now supported.

10. The logical operators || and && are supported. Testing proceeds left to right and ceases when the outcome is known.

11. The operators ~ and ! are now supported.

12. Local variables are now local to the block rather than the whole function.

13. The back-slash escape sequences for character and string constants have been added (per *DDJ* #56).

14. Some changes were made to accommodate the requirements and features of Small-VM and the assembler and loader which I use (Allen Ashley's PDS macro assembler and linking loader). They may be easily removed or modified to meet the requirements of your particular environment.

    a. Lower-case names are converted to upper case before being placed into the symbol table. Lower-case and upper-case symbols are thus synonymous. This feature is optional.

    b. Header and trailer code is automatically added to the output, enabling multi-file programs to be compiled and assembled separately (see *DDJ* #61 for details).

    c. Tabs are no longer generated in the output file. This feature is optional.

15. The original compiler would generate a cascade of spurious error messages, following a proper one. These have been eliminated by snuffing all but the first one in a simple statement; in practice this works well, since one diagnostic is usually enough to make one evaluate an entire simple statement.

16. The following fixes have been applied:

    a. The left- and right-shift routines of the arithmetic and logical library have been modified so that a shift of zero bits properly returns the original value (*DDJ* #52, pg. 7).

    b. Arguments no longer have to be typed in the same order as they appear in the function header (*DDJ* #52, pg. 20). A flaw in this patch has been corrected (*DDJ* #56, pg. 6).

    c. In its original form, Small-C did not always properly scale values involved in address arithmetic. *DDJ* #62 describes this problem and gives the solution.

    d. The function syname will no longer accept symbols longer than eight characters.

    e. The global integer lastst is now zeroed at the beginning of each function so null functions will always generate a return.

    f. Comments terminated by "*/" at the beginning of a line are now handled correctly (*DDJ* #56, pg. 6).

17. Literals are now dumped at the end of each function, permitting a smaller literal queue. Globals are defined at the point of their declaration, eliminating the function dumpglb.

18. An optional hash search may be used on the macro table and the global part of the symbol table. The improvement in speed is tremendous, but a 48K system will not simultaneously support this and all the new language features. So the new language statements and the hash search feature are compile-

time options, allowing users to pick and choose which features they prefer to support.

New Features Explained

The following comments describe some of the new features in greater detail:

1. Code Optimizing

Machine-independent optimizing was done by changing the expression analyzer, and machine-dependent optimizing was done by including an optional output (peephole) optimizer. The following techniques were used in the first case:

a. Expressions, or partial expressions, which result in a constant value generate only a single immediate load instruction.

b. After generating code for the right side of a binary operator, if the secondary register was not used, the precautionary push/pop of the left-side value is changed to a swap. But if the left-side value is a constant it is loaded directly into the secondary register instead.

c. Constants being added to or subtracted from integer pointers or array names are doubled at compile time rather than execution time.

d. No testing code is generated for "if(const)," "while(const)," and "for(. . .; const; . . .)" statements. The compiler does not bother to delete code controlled by these statements when const is zero since that situation is most likely a program error; the #ifdef and #ifndef should be used to eliminate conditional code at compile time.

e. Tests against zero (e.g., "while(i >= 0)," "if(abc() == 0)," etc.) result in special inline code which is smaller and faster than the standard procedure of loading zero, performing a library call, then testing the returned value for 1 or 0.

f. Zero subscripts generate no code for adding to the array address or pointer value.

g. Local variables are allocated all at one time when the first executable statement in a block is encountered. Declarations are not allowed after that point except within inner blocks.

h. Unnecessary jumps around statements controlled by an "else" are avoided. This is the case when a "return" or "goto" precedes the "else."

i. The function modstk now generates two swaps to preserve the primary register only on a "return" with an expression.

Machine-dependent optimization is done by two functions – putstk and peephole. Putstk now generates MOV A,L/STAX D rather than CALL CCPCHAR. Peephole is the output optimizer mentioned earlier. A staging buffer is used to hold the code generated by an expression. When the buffer is flushed, peephole scrutinizes the output, making changes as it sees fit. The output optimizer is a compile-time option since some may view this as something better suited to a separate utility program. I found it irresistible, however, to piggyback this simple function on the compiler; it makes for faster and easier optimized compiles.

Two techniques are employed by peephole. First, integers being fetched from the top of the stack are obtained with a POP H/PUSH H sequence rather than the usual LXI H,O/DAD SP/CALL CCGINT sequence. Integers next to the top of the stack are fetched with a POP B/POP H/PUSH H/PUSH B sequence. If an XCHG follows the fetch sequence then the desired operand is popped directly into the DE register pair. These techniques result in smaller, faster code; they are always effective if the compiler contains peephole.

The second technique involves replacing commonly occurring sequences of commands with calls to new entry points in the run-time library (Listing 3). This technique reduces program size at the expense of speed; it must be requested at run time to be effective.

2. Passing Argument Counts

When a function is called, a count of the number of arguments being passed is placed in the accumulator. This takes only two bytes. To fetch the count, the called function simply assigns to a variable the value returned by the function CCARGC (upper case). That must be done first in the function since other operations may call certain run-time library routines which destroy the accumulator. CCARGC is a new entry point in the run-time library; it simply redefines CCSXT which moves A to HL sign extended. That allows 127 arguments before going berserk. For obvious reasons, the compiler will not generate code to load an argument count for calls to CCARGC. Since many programs will not use the argument count feature, the compiler will skip it in programs containing the statement "#define NOCCARGC" (upper case). That reduces program size and run time.

3. Data Initialization

Global variables, array elements, and pointers may be initialized just as in full C, except that symbols may not be used as initializers. When not initialized, globals default to zero. An equal sign must introduce initializers.

Constant expressions may be used to initialize variables or array elements only. If the size of an array is not given, it is determined by the number of initializers present. Character constants with back-slash escape sequences are permitted. When multiple initializers are present, they must be enclosed in braces and separated by commas. If too few initializers are given, trailing elements are set to zero. If too many are given, an error message is issued.

A quoted character string may be used to initialize only chracter arrays and pointers. In this case, a terminating zero byte is automatically generated. An array name will reference the first byte and a pointer will contain the address of the first byte. If no array size is given, it will be set to the length of the string plus one. If the string is longer than the stated size, the size is increased to match the string.

One negative side effect of the zero default is that very large arrays generate many "DB 0,0,0,. . ." or DW 0,0,0,. . ." statements in the output. If this proves to be a problem in practice, a run-time option could be set up to sidestep the zero default.

4. Local Declarations

The original compiler would accept local declarations anywhere within a function; and duplicate declarations produced errors. This version requires local declarations to appear first within a block and permits multiple declarations of the same

symbol. The local part of the symbol table is now searched in reverse order so as to see the latest occurrence of a variable first. Upon leaving a block, the non-lable declarations occurring within it are stripped from the symbol table. Local declarations may not contain initializers.

5. Goto Statement

This statement was included with some reluctance. I see no compelling need for it, and I shudder to think of how easily its abuse camouflages the logic of a program. There are occasional situations, however, where it can prevent code redundancy without obscuring the logic. I included the "goto" largely because it is expected and also because it can prove invaluable when converting existing programs to Small-C.

There is one restriction in the use of the "goto" statement. Since local variables may be declared within any block, the compiler cannot know the level of the stack pointer at target labels which have not yet been defined; so it cannot adjust the stack pointer before the branch. Not being able to find an efficient method of solving this dilemma, I chose to make block-locals (other than at the start of a function) and "goto" statements mutually exclusive within a given function.

6. Extern Storage Class

The "extern" storage class may be specified with global declarations only. If the LINK option (see "Installation" below) is effective, such objects are defined as external references to the assembler and other globals are defined as entry points. If LINK is not effective, "extern" globals are not defined to the assembler and other globals are defined, but not as entry points. If "int" or "char" is not specified following "extern" then "int" is assumed.

The Standard I/O Library

Listing #2 is the source code for the standard I/O library which interfaces Small-C programs to Small-VM. These functions are used for ASCII character-stream operations. They were designed to mimic the full C standard I/O library functions as closely as possible. I strongly urge all Small-C implementors to follow suit.

This implementation uses a carriage-return/line-feed sequence to terminate lines, and a hex FF for end of file. Naturally, you will need to change these functions to work with your file format.

These functions, the arithmetic and logical library (Listing #3), and Small-VM are made known to Small-C programs by assembly language code found in either of the files stdio.h or stdiol.h (Listings #4 and #5). One of these must be selected for inclusion into a program at its beginning. If stdio is selected, the program will reference copies of these modules at fixed locations in memory. If stdiol is selected, then "local" copies of these modules will be loaded and linked together with the program. These files also contain "#define . . ." statements which give values for certain commonly used constants; e.g., EOF, ERR, stdin, etc.

A brief description of the standard I/O library functions follows:

- **fgetc(fd) int fd;** (alias getc)

Returns the next sequential character from the file indicated by fd (EOF for end-of-file or error). A null character or a hex FF (actually any value <= 0) indicates end-of-file. Control-D is also taken for end-of-file; this is the means of terminating keyboard input. If the character is a DEL it is changed to a backspace. If fd is an I/O port, the character is echoed to fd (or stderr if fd is stdin). Backspaces are not echoed. Incoming line-feed characters are ignored.

- **getchar()**

Same as fgetc(stdin).

- **fgets(string, size, fd) char *string; int size, fd;**

Reads up to size minus one characters into memory, starting at string, from file fd. A carriage-return terminates the input and is replaced with a null byte. Returns string for success, else NULL for end-of-file or an error. If fd is an I/O port, fgets obtains characters by calling fgetc which handles echoing and changes DEL characters to backspaces. Backspace characters cause a rubout of the previous character (if any) in the string; this echoes a backspace/space/backspace sequence. If fd is a disk file, a single call to CCREAD obtains the entire string. A call is first made to CCDELIM (a new entry point in Small-VM not reported in *DDJ* #61); this causes CCREAD to stop after transferring a carriage-return character. This is much faster than calling CCREAD for every character one at a time.

- **fputc(c, fd) char c; int fd;** (alias putc)

Writes the character, c, to the specified file. Returns the character itself on success, else EOF. If c is a carriage-return, then a line-feed is appended.

- **putchar(c) char c;**

Same as fputc(c, stdout).

- **fputs(string, fd) char *string; int fd;**

Writes characters beginning at string to the specified file until a NULL byte is encountered. The NULL is not written and a newline is not appended. Returns NULL for success, else EOF.

- **puts(string) char *string;**

Same as fputs(string, stdout), but appends a newline (carriage-return/line-feed sequence) to the output.

- **fopen(name, mode) char *name, *mode;**

Attempts to open the file indicated by the null-terminated character string at name. Mode points to a single-byte string indicating the open mode:

"r" – read only
"w" – write only
"u" – update (read/write)

If successful, fopen returns the fd value for the open file; otherwise, NULL.

- **fclose(fd) int fd;**

Closes the specified file. If the file is open for write only, and if it is a disk file, a hex FF (end-of-file flag) is written before closing.

- **fflush(fd) int fd;**

Forces any system-buffered changes out to the file. This function consists of a jump to CCFLUSH in Small-VM.

- **exit(errorcode) int errorcode;** (alias abort)

Closes all open files, displays errorcode on the console (if non-zero), and returns to DOS. This function consists of a jump to CCEXIT in Small-VM.

- **unlink(name) char *name;**

Deletes the named file. Returns NULL on success, else ERR. This function consists of a jump to CCPURGE in Small-VM.

- **getarg(argnbr, str, sz, argc, argv)**
  char *str; int argnbr, sz, argc, *argv;

Obtains the command-line argument indicated by argnbr, moves it to str of maximum size sz, and returns the size of the field obtained. Argc, and argv must be the same values provided to the function main when the program is started. If argnbr is zero, the program name is obtained. If it is one, the first argument following the program name is obtained and so on. If no parameter corresponds to argnbr, getarg sets str null and returns EOF.

The Auxiliary Library

The auxiliary library (Listing #6) is a collection of functions which augment the standard I/O library. They may be selected individually for inclusion into programs at compile time, or they may be loaded and linked as a group. Their descriptions follow:

- **abs(nbr) int nbr;**

Returns the absolute value of nbr.

- **dtoi(str, nbr) char *str; int *nbr;**

Converts the (possibly) signed decimal number in the character string at str to an integer in nbr and returns the length of the numeric field found. Dtoi stops when it finds the end of the string or any illegal numeric character. At most, a leading sign and five digits will be used.

- **itod(nbr, str, sz) int nbr, sz; char *str;**

Converts nbr to a signed (if negative) character string at str. Sz indicates the length of the string. The value is right justified and blank filled in str. The sign and possibly high-order digits are truncated if sz is too small. Returns str. If sz is greater than zero, a null byte is placed at str[sz-1]. If sz is zero, a search for the first null byte following str locates the end of the string. If sz is less than zero, all sz characters of str are used, including the last one.

- **itou(nbr, str, sz) int nbr, sz; char *str;**

Converts nbr to an unsigned decimal character string at str. Itou works just like itod except that the high-order bit or nbr is taken for a magnitude bit resulting in values from 0 to 65535.

- **itox(nbr, str, sz) int nbr, sz; char *str;**

Converts nbr to a hexadecimal character string at str. Sz indicates the length of the string. The result is right justified and blank filled in the destination string. High-order digits are truncated if sz is too small. Returns str. If sz is greater than zero, a null byte is placed at str[sz-1]. If sz is zero, a search for the first null byte following str locates the end of the string. If sz is less than zero, all sz characters of str are used, inluding the last one.

- **left(str) char *str;**

Left adjusts the character string at str. Starting with the first non-blank character and proceeding through the null terminator, the string is moved to address str. To obtain the left-adjusted ASCII character string representing the decimal value of an integer, code:

left(itod(nbr, str, sz))

- **printf(str, arg1, arg2, . . ., argn) char *str;**

Converts the arguments (arg1, arg2, . . ., argn) to character strings according to specifications contained in the control string at str. Str is required, but arg1 through argn are optional. Printf operates exactly as described in *The C Programming Language* except that only the d, x, c, s, and u specifications are supported.

- **sign(nbr) int nbr;**

Returns -1, 0 or +1 depending on whether nbr is less than, equal to, or greater than zero.

- **strcmp(str1, str2) char *str1, *str2;**

Returns -1, 0 or +1 depending on whether the string at str1 is less than, equal to, or greater than the string at str2. Character-by-character comparisons are made from left to right according to the numeric value of the character codes. The first unequal match terminates the compare. Only if every character in str1 matches the corresponding character in str2 is a zero returned. Str1 is considered greater than str2 if str2 is equal but shorter than str1 and vice versa.

- **utoi(str, nbr) char *str; int *nbr;**

Converts the unsigned decimal number represented by the character string at str to an integer in nbr and returns the length of the numeric field found in str. Utoi stops when it encounters the end of the string or any nondecimal character. Five digits at most will be used.

- **xtoi(str, nbr) char *str; int *nbr;**

Converts the hexadecimal number in the character string at str to an integer in nbr and returns the length of the hexadecimal field found in str. Xtoi stops when it encounters a non-hexadecimal digit in str. Four digits at most will be used.

Invoking The Compiler

Two methods of obtaining run-time parameters are provided. If your environment supports UNIX-like command line argument passing, then the needed parameters may be picked off the command line. In this case, three types of parameters may be given – file names, redirection specifications, and switches. The operating system should handle redirection specifications without passing them on to the program.

By default, Small-C obtains its input from the standard input file (stdin). If a list of file names is given on the command line, then instead of reading stdin, it reads the named files in the order listed. Any non-switch parameter is taken for a file name. Output always goes to the standard output file (stdout).

Switches are preceded by a hyphen. The "-m" switch allows one to *monitor* progress by having the compiler write

(Continued on page 31)

Small-C Compiler, v.2 (Text begins on page 16)

Listing One (Part one)

Due to its length, this listing will be continued next month.

```
   File: cc.def,2

 1 /*
 2 ** Small-C Compiler Version 2.0
 3 **
 4 ** Copyright 1982 J. E. Hendrix
 5 **
 6 ** Macro Definitions
 7 */
 8
 9 /*
10 ** compile options
11 */
12 #define PHASE2    /* 2nd and later compiles */
13 #define SEPARATE /* compile separately */
14 #define OPTIMIZE /* compile output optimizer */
15 #define NOCCARGC /* no calls to CCARGC */
16 #define HASH     /* use hash search for macros */
17 #define SMALL_VM /* uses Small-VM interface */
18 #define CMD_LINE /* command line run options */
19 #define DYNAMIC  /* allocate memory dynamically */
20 #define POLL     /* poll for operator interruptions */
21 #define PDS      /* uses PDS assembler and loader */
22 #define COL      /* terminate labels with a colon */
23 /* #define TAB  9    /* put out tabs of this value */
24 #define UPPER    /* force symbols to upper case */
25 #define LINK     /* will use with linking loader */
26
27 /*
28 ** machine dependent parameters
29 */
30 #define BPW      2   /* bytes per word */
31 #define LBPW     1   /* log2(BPW) */
32 #define SBPC     1   /* stack bytes per character */
33 #define ERRCODE  7   /* op sys return code */
34
35 /*
36 ** symbol table format
37 */
38 #define IDENT    0
39 #define TYPE     1
40 #define CLASS    2
41 #define OFFSET   3
42 #define NAME     5
43 #define OFFSIZE (NAME-OFFSET)
44 #define SYMAVG   10
45 #define SYMMAX   14
46
47 /*
48 ** symbol table parameters
49 */
50 #define NUMLOCS   25
51 #define STARTLOC  symtab
52 #define ENDLOC   (symtab+(NUMLOCS*SYMAVG))
53 #define NUMGLBS   180
54 #define STARTGLB  ENDLOC
55 #define ENDGLB   (ENDLOC+((NUMGLBS-1)*SYMMAX))
56 #define SYMTBSZ   2770  /* NUMLOCS*SYMAVG + NUMGLBS*SYMMAX */
57
58 /*
59 ** input line
60 */
61 #define LINEMAX  80
62 #define LINESIZE 81
63
64 /*
65 ** output staging buffer size
66 */
67 #define STAGESIZE   800
68 #define STAGELIMIT  (STAGESIZE-1)
69
70 /*
71 ** macro (define) pool
72 */
73 #ifdef HASH
74 #define MACNBR   90
75 #define MACNSIZE 990   /* 90*(NAMESIZE+2) */
76 #define MACNEND  (macn+MACNSIZE)
77 #define MACQSIZE 450   /* 90*5 */
78 #else
79 #define MACQSIZE 950
80 #endif
81 #define MACMAX  (MACQSIZE-1)
82
83 /*
84 ** statement types
85 */
86 #define STIF      1
87 #define STWHILE   2
88 #define STRETURN  3
89 #define STBREAK   4
90 #define STCONT    5
91 #define STASM     6
92 #define STEXPR    7
93 #define STDO      8 /* compile "do" logic */
94 #define STFOR     9 /* compile "for" logic */
95 #define STSWITCH 10 /* compile "switch/case/default" logic */
96 #define STCASE   11
97 #define STDEF    12
98 #define STGOTO   13 /* compile "goto" logic */
99
```

(Part two)

```
   File: cc1.c,2

 1 /*
 2 ** Small-C Compiler Version 2.0
 3 **
 4 ** Copyright 1982 J. E. Hendrix
 5 **
 6 ** Part 1
 7 */
 8 #include stdio.h,2
 9 #include cc.def,2
10
11 /*
12 ** miscellaneous storage
13 */
14 char
15 #ifdef OPTIMIZE
16   optimize, /* optimize output of staging buffer */
17 #endif
18   alarm,    /* audible alarm on errors? */
```

```
57
58 /*
59 ** System wide name size (for symbols)
60 */
61 #define NAMESIZE 9
62 #define NAMEMAX  8
63
64 /*
65 ** possible entries for "IDENT"
66 */
67 #define LABEL    0
68 #define VARIABLE 1
69 #define ARRAY    2
70 #define POINTER  3
71 #define FUNCTION 4
72
73 /*
74 ** possible entries for "TYPE"
75 **    low order 2 bits make type unique within length
76 **    high order bits give length of object
77 */
78 /*      LABEL   0 */
79 #define CCHAR   (1<<2)
80 #define CINT    (BPW<<2)
81
82 /*
83 ** possible entries for "CLASS"
84 */
85 /*      LABEL     0 */
86 #define STATIC    1
87 #define AUTOMATIC 2
88 #define EXTERNAL  3
89
90 /*
91 ** "switch" table
92 */
93
94 #ifdef PHASE2
95 #define SWSIZ   (2*BPW)
96 #define SWTABSZ (25*SWSIZ)
97 #else
98 #define SWSIZ    4
99 #define SWTABSZ 100
100 #endif
101
102 /*
103 ** "while" statement queue
104 */
105 #define WQTABSZ  30
106 #define WQSIZ     3
107 #define WQMAX   (wq+WQTABSZ-WQSIZ)
108
109 /*
110 ** entry offsets in while queue
111 */
112 #define WQSP    0
113 #define WQLOOP  1
114 #define WQEXIT  2
115
116 /*

117 ** literal pool
118 */
119 #define LITABSZ 700
120 #define LITMAX  (LITABSZ-1)
```

(Continued on top of page 28)

```
19   monitor,  /* monitor function headers? */
20   pause,    /* pause for operator on errors? */
21 #ifdef DYNAMIC
22  *stage,    /* output staging buffer */
23  *symtab,   /* symbol table */
24  *litq,     /* literal pool */
25 #ifdef HASH
26  *macn,     /* macro name buffer */
27 #endif
28  *macq,     /* macro string buffer */
29  *pline,    /* parsing buffer */
30  *mline,    /* macro buffer */
31 #else
32   stage[STAGESIZE],
33   symtab[SYMTBSZ],
34   litq[LITABSZ],
35 #ifdef HASH
36   macn[MACNSIZE],
37 #endif
38   macq[MACQSIZE],
39   pline[LINESIZE],
40   mline[LINESIZE],
41   swq[SWTABSZ],
42 #endif
43  *line,     /* points to pline or mline */
44  *lptr,     /* ptr to either */
45  *glbptr,   /* ptrs to next entries */
46  *locptr,   /* ptr to next local symbol */
47  *stagenext,/* next addr in stage */
48  *stagelast,/* last addr in stage */
49   quote[2], /* literal string for '"' */
50  *cptr,     /* work ptrs to any char buffer */
51  *cptr2,
52  *cptr3,
53   msname[NAMESIZE], /* macro symbol name array */
54   ssname[NAMESIZE]; /* static symbol name array */
55 int
56 #ifdef STGOTO
57   nogo,     /* > 0 disables goto statements */
58   noloc,    /* > 0 disables block locals */

59 #endif
60   op[16],   /* function addresses of binary operators */
61   op2[16],  /* same for unsigned operators */
62   opindex,  /* index to matched operator */
63   opsize,   /* size of operator in bytes */
64   swactive, /* true inside a switch */
65   swdefault,/* default label #, else 0 */
66  *swnext,   /* address of next entry */
67  *swend,    /* address of last table entry */
68 #ifdef DYNAMIC
69  *wq,       /* while queue */
70 #else
71   wq[WQTABSZ],
72 #endif
73 #ifdef CMD_LINE
74   argcs,    /* static argc */
75  *argvs,    /* static argv */
76 #endif
77  *wqptr,    /* ptr to next entry */
78   litptr,   /* ptr to next entry */
79   macptr,   /* macro buffer index */
80 #ifndef HASH
81   mack,     /* variable k for findmac routine */
82 #endif
```

(Continued on next page)

Small-C Compiler, v.2

(Listing continued, text begins on page 16)

Listing One (Part two continued)

```
 83   pptr,      /* ptr to parsing buffer */
 84   oper,      /* address of binary operator function */
 85   ch,        /* current character of line being scanned */
 86   nch,       /* next character of line being scanned */
 87   declared,  /* # of local bytes declared, else -1 when done */
 88   iflevel,   /* #if... nest level */
 89   skiplevel,/* level at which #if... skipping started */
 90   func1,     /* true for first function */
 91   nxtlab,    /* next avail label # */
 92   litlab,    /* label # assigned to literal pool */
 93   beglab,    /* beginning label -- first function */
 94   csp,       /* compiler relative stk ptr */
 95   argstk,    /* function arg sp */
 96   argtop,
 97   ncmp,      /* # open compound statements */
 98   errflag,   /* non-zero after 1st error in statement */
 99   eof,       /* set non-zero on final input eof */
100   input,     /* fd # for input file */
101   input2,    /* fd # for "include" file */
102   output,    /* fd # for output file */
103   files,     /* non-zero if file list specified on cmd line */
104   filearg,   /* cur file arg index */
105   glbflag,   /* non-zero if internal globals */
106   ctext,     /* non-zero to intermix c-source */
107   ccode,     /* non-zero while parsing c-code */
108              /* zero when passing assembly code */
109   listfp,    /* file pointer to list device */
110   laststt,   /* last executed statement type */
111  *iptr;      /* work ptr to any int buffer */
112
113 #ifdef SEPARATE
114 #ifdef LINK
115 #ifdef PDS
116 #asm

117 CC2.R:LIBRY
118 CC3.R:LIBRY
119 CC4.R:LIBRY
120 #endasm
121 #endif
122 #endif
123 extern int
124   addmac(), addsym(), addwhile(), amatch(), blanks(),
125   bump(), clearstage(), col(), delwhile(), endst(),
126   error(), findglb(), findloc(), gch(), getint(),
127   getlabel(), illname(), inbyte(), inline(), junk(),
128   kill(), lout(), match(), multidef(), needtoken(),
129   nextsym(), nl(), numeric(), outbyte(), outdec(),
130   postlabel(), preprocess(), printlabel(), putint(),
131   readwhile(), setstage(), sout(), streq(), symname(),
132   upper();
133
134 extern int
135   constexpr(), expression(), number(), qstr(),
136   test(), stowlit();
137
138 extern int
139   add(), and(), asl(), asr(), defstora(),
140   div(), eq(), entry(), external(), ge(),
141   gt(), header(), jump(), le(), lt(), mod(), modstk(),
142   mult(), ne(), or(), point(), ret(), sub(), sw(),
143   trailer(), uge(), ugt(), ule(), ult(), xor();
144 #endif
145
146 #include cc11.c,2
147 #include cc12.c,2
148 #include cc13.c,2
149
150 #ifndef SEPARATE
151 #include cc21.c,2
152 #include cc22.c,2
153 #include cc31.c,2
154 #include cc32.c,2
155 #include cc33.c,2
156 #include cc41.c,2
157 #include cc42.c,2
158 #endif
```

```
  1 /*
  2 ** execution begins here
  3 */
  4 #ifdef CMD_LINE
  5 main(argc, argv) int argc, *argv; {
  6   argcs=argc;
  7   argvs=argv;
  8 #else
  9 main() {
 10 #endif
 11 #ifdef DYNAMIC
 12   swnext=CCALLOC(SWTABSZ);
 13   swend=swnext+((SWTABSZ-SWSIZ)>>1);
 14   stage=CCALLOC(STAGESIZE);
 15   stagelast=stage+STAGELIMIT;
 16   wq=CCALLOC(WQTABSZ*BPW);
 17   litq=CCALLOC(LITABSZ);
 18 #ifdef HASH
 19   macn=CCALLOC(MACNSIZE);
 20   cptr=macn-1;
 21   while(++cptr < MACNEND) *cptr=0;
 22 #endif
 23   macq=CCALLOC(MACQSIZE);
 24   pline=CCALLOC(LINESIZE);
 25   mline=CCALLOC(LINESIZE);
 26 #else
 27   swend=(swnext=swq)+SWTABSZ-SWSIZ;
 28   stagelast=stage+STAGELIMIT;
 29 #endif
 30   swactive=        /* not in switch */
 31   stagenext=       /* direct output mode */
 32   iflevel=         /* #if... nesting level = 0 */
 33   skiplevel=       /* #if... not encountered */
 34   macptr=          /* clear the macro pool */
 35   csp =            /* stack ptr (relative) */
 36   errflag=         /* not skipping errors till ";" */
 37   eof=             /* not eof yet */
 38   ncmp=            /* not in compound statement */
 39   files=
 40   filearg=
 41   quote[1]=0;
 42   func1=           /* first function */
 43   ccode=1;         /* enable preprocessing */
 44   wqptr=wq;        /* clear while queue */
 45   quote[0]='"';    /* fake a quote literal */
 46   input=input2=EOF;
 47   ask();           /* get user options */
 48   openin();        /* and initial input file */
```

(Continued on column 1, page 34)

(Continued from page 22)

each function header line to the console. This switch is useful for isolating errors to the functions containing them.

The switch "-a" causes the *alarm* (control-G to the console) to sound when an error is reported.

The switch "-p" causes the compiler to *pause* after reporting each error. A carriage-return from the keyboard continues processing.

The switch "-l#" (# is a file descriptor in the range 1-9) instructs Small-C to *list* the source code on the file indicated. If file descriptor 1 (stdout) is specified, the listing is mixed with the normal output. In this case, a semicolon precedes each line of source code. No listing is produced if the switch is not given.

Machine-independent and pop/push optimizing (described earlier) are always performed. The switch "-o" causes the output *optimizer* to further reduce program size at the expense of execution speed.

The "-b#" switch exists only if the compiler will not be used with a linking loader (see "Installation" below); i.e., program parts will be combined at assembly time rather than load time. It causes label numbering to *begin* following the value #. If # is zero (default) a complete program is being compiled. In this case, header and trailer code, designed to link the program with its environment, is appended to the program. A value of 1 means the first of a multi-part program is being compiled; header code only will be appended. A value between 1 and 9000 identifies an intermediate part; in this case no code is appended to the output. A value of 9000 means the last part is being compiled; trailer code only will be appended. Values for # must be chosen to prevent clashes with labels generated in other parts of the program.

The null switch "-" or any undefined switch causes the compiler to exit after displaying the help line:

```
usage: cc [file] . . . [-m] [-a] [-p] [-l#] [-o] [-b#]
```

The alternative to command-line parameters is prompting. The operator is asked for the output file name and then for each of the run-time options. The final prompt is for an input file. At the end of that file, another input file is sought. A null response terminates the compile.

Installation

This version of Small-C may be compiled using the original compiler. First, however, you will need to make the following changes to your present compiler (if not already done).

1. Apply the fixes mentioned earlier (item 16 under "Differences"). Fixes (c), (d), and (e) may be omitted.

2. Install the change which allows the compiler to properly handle back-slash escape sequences (*DDJ* #56, pg. 6).

3. Make whatever changes are required by your run-time environment. Be sure your system supports the Standard I/O Library functions "fgets," "fputs," and "fputc" (Listing #2). All input and output is through these functions.

> **Note**: The original compiler, in several instances, performed a "logical and" of 127 with a source byte. I have removed these operations preferring to let the input routine do it (if required at all).

Then you should prepare a temporary copy of the new compiler as input to the first (using your present updated compiler) of two compiles. This is done by deleting lines from the new compiler which cannot be handled by the present one or which are not desirable for your particular installation.

> **Note**: If you only have 48K of memory, you probably will not be able to support all the new language statements, the code optimizer, and the hash search algorithm. I recommend dropping all of these except the optimizer at first, then adding features until your limit is reached. The new expression analyzer uses much more stack space than the old one, and each time embedded parentheses are encountered, the entire expression hierarchy must be traversed again from top to bottom, requiring about 400 more bytes. If you bomb off on such a statement, for lack of memory, try breaking down the statement into intermediate stages.

Conditional compilation statements (#ifdef, #ifndef, #else, and #endif) are present in the source files to document which lines should be deleted and which ones should be retained. For the first compile, they must be deleted along with all inappropriate lines. Once a usable compiler has been generated, the temporary source files should be discarded and the newly generated compiler must be used to produce a second version from the full source files. This second compile (phase 2) is required because it will include statements which could not be compiled by the original compiler (phase 1).

Several symbols are defined in the file "cc.def" for the purpose of controlling compiler options. If the action taken by the compiler is inappropriate when any of these symbols is defined, that symbol should be deleted or commented out. These symbols are described below:

SMALL_VM compiles statements in the file cc41.c which cause header and trailer code appropriate for the Small-VM interface to be generated for each program compiled.

> **Note**: Small-VM entry points are called at a number of places in the compiler. They permit dynamic memory allocation and operator interruption/resumption of compiler execution. All Small-VM calls are coded in upper case and begin with a "CC" prefix. For a description of these functions see *DDJ* #61. If your run-time system supports either or both of these features, then you should adapt the applicable ones and discard the others.

DYNAMIC compiles statements which dynamically allocate memory for various tables and arrays within the compiler. If DYNAMIC is not defined, the tables and arrays are compiled directly into the compiler. This symbol also controls statements which call CCAVAIL whose primary purpose is to return the amount of free memory remaining, but is used here to verify that the machine stack and allocated memory do not overlap; the run aborts in that case. If serial table searching is used with dynamic memory allocation, then each new entry to the global symbol table is allocated separately. The table may grow until it overlaps the machine stack, producing an allocation error.

(Continued on page 56)

Small-C Compiler, v.2

(Listing continued, text begins on page 16)

Listing One (Part two continued)

```
49    preprocess();    /* fetch first line */
50  #ifdef SMALL_VM
51    fopen(" ",NULL);/* pre-alloc FCB for include file */
52  #endif
53  #ifdef DYNAMIC
54  #ifdef HASH
55    symtab=CCALLOC(NUMLOCS*SYMAVG + NUMGLBS*SYMMAX);
56  #else
57    symtab=CCALLOC(NUMLOCS*SYMAVG);
58    /*  global space is allocated with each new entry  */

59  #endif
60  #endif
61  #ifdef HASH
62    cptr=STARTGLB-1;
63    while(++cptr < ENDGLB) *cptr=0;
64  #endif
65    glbptr=STARTGLB;
66    glbflag=1;
67    ctext=0;
68    header();           /* intro code */
69    setops();           /* set values in op arrays */
70    parse();            /* process ALL input */
71    outside();          /* verify outside any function */
72    trailer();          /* follow-up code */
73    fclose(output);
74    }
75
76  /*
77  ** process all input text
78  **
79  ** At this level, only static declarations,
80  **      defines, includes and function
81  **      definitions are legal...
82  */
83  parse() {
84    while (eof==0) {
85      if(amatch("extern", 6))    dodeclare(EXTERNAL);
86      else if(dodeclare(STATIC));
87      else if(match("#asm"))     doasm();
88      else if(match("#include"))doinclude();
89      else if(match("#define")) addmac();
90      else                       newfunc();
91      blanks();       /* force eof if pending */
92      }
93    }
94
95  /*
96  ** dump the literal pool
97  */
98  dumplits(size) int size; {
99    int j, k;
100   k=0;
101   while (k<litptr) {
102 #ifdef POLL
103     CCPOLL(1); /* allow program interruption */
104 #endif
105     defstorage(size);
106     j=10;
107     while(j--) {
173         monitor=YES;
174         continue;
175         }
176 #ifdef OPTIMIZE
177       if(upper(line[1])=='O') {
178         optimize=YES;
179         continue;
180         }
181 #endif
182       if(upper(line[1])=='P') {
183         pause=YES;
184         continue;
185         }
186       }
187 #ifndef LINK
188     if(upper(line[1])=='B') {
189       bump(0); bump(2);
190       if(number(&nxtlab)) continue;
191       }
192 #endif
193     sout("usage: cc [file]... [-m] [-a] [-p] [-l#]", stderr);
194 #ifdef OPTIMIZE
195     sout(" [-o]", stderr);
196 #endif
197 #ifndef LINK
198     sout(" [-b#]", stderr);
199 #endif
200     sout("\n", stderr);
201     abort(ERRCODE);
202     }
203   }
204
205 #else
206
207 ask() {
208 #ifdef OPTIMIZE
209   optimize=
210 #endif
211   monitor=alarm=pause=listfp=nxtlab=0;
212   line=mline;
213   while(1) {
214     prompt("Output file: ", line, LINESIZE);
215     if(output=fopen(line, "w")) break;
216     else lout("open error");
217     }
218 #ifndef LINK
219   while(1) {
220     prompt("Beginning label number: ", line, LINESIZE);
221     bump(0);
222     if(number(&nxtlab)) break;
223     }
224 #endif
225   while(1) {
226     prompt("Monitor function headers? ", line, LINESIZE);
227     if(upper(*line)=='Y') monitor=YES;
228     else if(upper(*line)!='N') continue;
229     break;
230     }
231   while(1) {
232     prompt("Sound alarm on errors? ", line, LINESIZE);
233     if(upper(*line)=='Y') alarm=YES;
234     else if(upper(*line)!='N') continue;
235     break;
236     }
237   while(1) {
238     prompt("Pause on errors? ", line, LINESIZE);
239     if(upper(*line)=='Y') pause=YES;
240     else if(upper(*line)!='N') continue;
```

```
108       outdec(getint(litq+k, size));
109       k=k+size;
110       if ((j==0);(k>=litptr)) {
111         nl();
112         break;
113         }
114       outbyte(',');
115       }
116     }

117   }
118
119 /*
120 ** dump zeroes for default initial values
121 */
122 dumpzero(size, count) int size, count; {
123   int j;
124   while (count > 0) {
125 #ifdef POLL
126     CCPOLL(1); /* allow program interruption */
127 #endif
128     defstorage(size);
129     j=30;
130     while(j--) {
131       outdec(0);
132       if ((--count <= 0);(j==0)) {
133         nl();
134         break;
135         }
136       outbyte(',');
137       }
138     }
139   }
140
141 /*
142 ** verify compile ends outside any function
143 */
144 outside()  {
145   if (ncmp) error("no closing bracket");
146   }
147
148 /*
149 ** get run options
150 */
151 #ifdef CMD_LINE
152 ask() {
153   int i;
154   i=listfp=nxtlab=0;
155   output=stdout;
156 #ifdef OPTIMIZE
157   optimize=
158 #endif
159   alarm=monitor=pause=NO;
160   line=mline;
161   while(getarg(++i, line, LINESIZE, argcs, argvs)!=EOF) {
162     if(line[0]!='-') continue;
163     if((upper(line[1])=='L')&(numeric(line[2]))&(line[3]<=' ')) {
164       listfp=line[2]-'0';
165       continue;
166       }
167     if(line[2]<=' ') {
168       if(upper(line[1])=='A') {
169         alarm=YES;
170         continue;
171         }
172       if(upper(line[1])=='M') {
```

(Continued on top of page 34)

```
241       break;
242       }
243 #ifdef OPTIMIZE
244   while(1) {
245     prompt("Optimize for size? ", line, LINESIZE);
246     if(upper(*line)=='Y') optimize=YES;
247     else if(upper(*line)!='N') continue;
248     break;
249     }
250 #endif
251   while(1) {
252     prompt("Listing file descriptor: ", line, LINESIZE);
253     if(numeric(*line)&(line[1]==NULL)) listfp=*line-'0';
254     else if(*line!=NULL) continue;
255     break;
256     }
257   }
258 #endif
259
260 /*
261 ** get next input file
262 */
263 openin() {
264   input=EOF;
265 #ifdef CMD_LINE
266   while(getarg(++filearg, pline, LINESIZE, argcs, argvs)!=EOF) {
267     if(pline[0]=='-') continue;
268 #else
269   while(prompt("Input file: ", pline, LINESIZE)) {
270 #endif
271     if((input=fopen(pline,"r"))==NULL) {
272       lout("open error", stderr);
273       abort(ERRCODE);
274       }
275     files=YES;
276     kill();
277     return;
278     }
279   if(files++) eof=YES;
280   else input=stdin;
281   kill();
282   }
283 #ifndef CMD_LINE
284
285 prompt(msg, ans, anslen) char *msg, *ans; int anslen; {
286   sout(msg, stderr);
287   fgets(ans, anslen, stderr);
288   }
289 #endif
290

291 setops() {
292   op2[00]=     op[00]=  or;  /* heir5 */
293   op2[01]=     op[01]= xor;  /* heir6 */
294   op2[02]=     op[02]= and;  /* heir7 */
295   op2[03]=     op[03]=  eq;  /* heir8 */
296   op2[04]=     op[04]=  ne;
297   op2[05]=ule; op[05]=  le;  /* heir9 */
298   op2[06]=uge; op[06]=  ge;
299   op2[07]=ult; op[07]=  lt;
300   op2[08]=ugt; op[08]=  gt;
301   op2[09]=     op[09]= asr;  /* heir10 */
302   op2[10]=     op[10]= asl;
303   op2[11]=     op[11]= add;  /* heir11 */
304   op2[12]=     op[12]= sub;
305   op2[13]=     op[13]=mult;  /* heir12 */
```

(Continued on next page)

Small-C Compiler, v.2

(Listing continued, text begins on page 16)

Listing One (Part two continued)

```
306   op2[14]=      op[14]= div;
307   op2[15]=      op[15]= mod;
308   )
309
```

```
  1 /*
  2 ** open an include file
  3 */
  4 doinclude()  {
  5   blanks();         /* skip over to name */
  6   if((input2=fopen(lptr,"r"))==NULL) {
  7     input2=EOF;
  8     error("open failure on include file");
  9     }
 10   kill();           /* clear rest of line */
 11        /* so next read will come from */
 12        /* new file (if open */
 13   }
 14
 15 /*
 16 ** test for global declarations
 17 */
 18 dodeclare(class) int class; {
 19   if(amatch("char",4)) {
 20     declglb(CCHAR, class);
 21     ns();
 22     return 1;
 23     }
 24   else if((amatch("int",3));(class==EXTERNAL)) {
 25     declglb(CINT, class);
 26     ns();
 27     return 1;
 28     }
 29   return 0;
 30   }
 31
 32 /*
 33 ** delcare a static variable
 34 */
 35 declglb(type, class)  int type, class; {
 36   int k, j;
 37   while(1) {
 38     if(endst()) return;      /* do line */
 39     if(match("*")) {
 40       j=POINTER;
 41       k=0;
 42       }
 43     else {
 44       j=VARIABLE;
 45       k=1;
 46       }
 47     if (symname(ssname, YES)==0) illname();
 48     if(findglb(ssname)) multidef(ssname);
 49     if(match("()")) j=FUNCTION;
 50     else if (match("[")) {
 51       k=needsub();    /* get size */
 52       j=ARRAY;   /* !0=array */
 53       }
 54     if(class==EXTERNAL) external(ssname);
 55     else j=initials(type>>2, j, k);
```

```
121   return ident;
122   }
123
124 /*
125 ** evaluate one initializer
126 */
127 init(size, ident, dim) int size, ident, *dim; {
128   int value;
129   if(qstr(&value)) {
130     if((ident==VARIABLE);(size!=1))
131       error("must assign to char pointer or array");
132     *dim = *dim - (litptr - value);
133     if(ident==POINTER) point();
134     }
135   else if(constexpr(&value)) {
136     if(ident==POINTER) error("cannot assign to pointer");
137     stowlit(value, size);
138     *dim = *dim - 1;
139     }
140   }
141
142 /*
143 ** get required array size
144 */
145 needsub()  {
146   int val;
147   if(match("]")) return 0; /* null size */
148   if (constexpr(&val)==0) val=1;
149   if (val<0) {
150     error("negative size illegal");
151     val = -val;
152     }
153   needtoken("]");       /* force single dimension */
154   return val;           /* and return size */
155   }
156
157 /*
158 ** begin a function
159 **
160 ** called from "parse" and tries to make a function
161 ** out of the following text
162 **
163 ** Patched per P.L. Woods (DDJ #52)
164 */
165 newfunc()  {
166   char *ptr;
167 #ifdef STGOTO
168   nogo  =               /* enable goto statements */
169   noloc = 0;            /* enable block-local declarations */
170 #endif
171   laststst=             /* no statement yet */
172   litptr=0;             /* clear lit pool */
173   litlab=getlabel();    /* label next lit pool */
174   locptr=STARTLOC;      /* clear local variables */
175   if(monitor) lout(line, stderr);
176   if (symname(ssname, YES)==0) {
177     error("illegal function or declaration");
178     kill(); /* invalidate line */
179     return;
180     }
181   if(func1) {
182     postlabel(beglab);
183     func1=0;
184     }
185   if(ptr=findglb(ssname)) {      /* already in symbol table ? */
186     if(ptr[IDENT]!=FUNCTION)       multidef(ssname);
187     else if(ptr[OFFSET]==FUNCTION) multidef(ssname);
188     else ptr[OFFSET]=FUNCTION;
```

```
    addsym(ssname, j, type, k, &glbptr, class);
    if (match(",")==0) return; /* more? */
    )
  )

/*
** declare local variables
*/
declloc(typ)  int typ; {
  int k,j;
#ifdef STGOTO
  if(noloc) error("not allowed with goto");
#endif
  if(declared < 0) error("must declare first in block");
  while(1) {
    while(1) {
      if(endst()) return;
      if(match("*")) j=POINTER;
      else j=VARIABLE;
      if (symname(ssname, YES)==0) illname();
      /* no multidef check, block-locals are together */
      k=BPW;
      if (match("[")) {
        k=needsub();
        if(k) {
          j=ARRAY;
          if(typ==CINT)k=k<<LBPW;
          }
        else j=POINTER;
        }
      else if(match("()")) j=FUNCTION;
      else if((typ==CCHAR)&(j==VARIABLE)) k=SBPC;
      declared = declared + k;
      addsym(ssname, j, typ, csp - declared, &locptr, AUTOMATIC);
      break;
      }
    if (match(",")==0) return;
    )
  }

/*
** initialize global objects
*/
initials(size, ident, dim) int size, ident, dim; {
  int savedim;
  litptr=0;
  if(dim==0) dim = -1;
  savedim=dim;
  entry();
  if(match("=")) {
    if(match("{")) {
      while(dim) {
        init(size, ident, &dim);
        if(match(",")==0) break;
        }
      needtoken("}");
      }
    else init(size, ident, &dim);
    }
  if((dim == -1)&(dim==savedim)) {
     stowlit(0, size=BPW);
    ident=POINTER;
    }
  dumplits(size);
  dumpzero(size, dim);
```

(Continued on top of page 36)

```
      /*  earlier assumed to be a function */
    }
  else
    addsym(ssname, FUNCTION, CINT, FUNCTION, &glbptr, STATIC);
  if(match("(")==0) error("no open paren");
  entry();
  locptr=STARTLOC;
  argstk=0;                /* init arg count */
  while(match(")")==0) {  /* then count args */
    /* any legal name bumps arg count */
    if(symname(ssname, YES)) {
      if(findloc(ssname)) multidef(ssname);
      else {
        addsym(ssname, 0, 0, argstk, &locptr, AUTOMATIC);
        argstk=argstk+BPW;
        }
      }
    else {error("illegal argument name");junk();}
    blanks();
    /* if not closing paren, should be comma */
    if(streq(lptr,")")==0) {
      if(match(",")==0) error("no comma");
      }
    if(endst()) break;
    }
  csp=0;        /* preset stack ptr */
  argtop=argstk;
  while(argstk) {
    /* now let user declare what types of things */
    /*      those arguments were */
    if(amatch("char",4))     {doargs(CCHAR);ns();}
    else if(amatch("int",3)) {doargs(CINT);ns();}
    else {error("wrong number of arguments");break;}
    }
  if(statement()!=STRETURN) ret();
  if(litptr) {
    printlabel(litlab);
    col();
    dumplits(1); /* dump literals */
    }
  }

/*
** declare argument types
**
** called from "newfunc" this routine adds an entry in the
** local symbol table for each named argument
**
** rewritten per P.L. Woods (DDJ #52)
*/
doargs(t) int t; {
  int j, legalname;
  char c, *argptr;
  while(1) {
    if(argstk==0) return; /* no arguments */
    if(match("*")) j=POINTER; else j=VARIABLE;
    if((legalname=symname(ssname, YES))==0) illname();
    if(match("[")) {   /* is it a pointer? */
      /* yes, so skip stuff between "[...]" */
      while(inbyte()!=']') if(endst()) break;
      j=POINTER; /* add entry as pointer */
      }
    if(legalname) {
      if(argptr=findloc(ssname)) {
        /* add details of type and address */
```

(Continued on next page

Small-C Compiler, v.2

(Listing continued, text begins on page 16)

Listing One (Part two continued)

```
254         argptr[IDENT]=j;
255         argptr[TYPE]=t;
256         putint(argtop-getint(argptr+OFFSET, OFFSIZE), argptr+OFFSET, OFFSIZE);
257         }
258       else error("not an argument");
259       }
260     argstk=argstk-BPW;        /* cnt down */
261     if(endst())return;
262     if(match(",")==0) error("no comma");
263     }
264   )
265
```

```
  1 /*
  2 ** statement parser
  3 **
  4 ** called whenever syntax requires a statement
  5 **  this routine performs that statement
  6 **  and returns a number telling which one
  7 */
  8 statement() {
  9   if ((ch==0) & (eof)) return;
 10   else if(amatch("char",4))  (declloc(CCHAR);ns();)
 11   else if(amatch("int",3))   (declloc(CINT);ns();)
 12   else {
 13     if(declared >= 0) {
 14 #ifdef STGOTO
 15       if(ncmp > 1) nogo=declared; /* disable goto if any */
 16 #endif
 17       csp=modstk(csp - declared, NO);
 18       declared = -1;
 19       }
 20     if(match("{"))                compound();
 21     else if(amatch("if",2))       (doif();laststst=STIF;)
 22     else if(amatch("while",5))    (dowhile();laststst=STWHILE;)
 23 #ifdef STDO
 24     else if(amatch("do",2))       (dodo();laststst=STDO;)
 25 #endif
 26 #ifdef STFOR
 27     else if(amatch("for",3))      (dofor();laststst=STFOR;)
 28 #endif
 29 #ifdef STSWITCH
 30     else if(amatch("switch",6))   (doswitch();laststst=STSWITCH;)
 31     else if(amatch("case",4))     (docase();laststst=STCASE;)
 32     else if(amatch("default",7)) (dodefault();laststst=STDEF;)
 33 #endif
 34 #ifdef STGOTO
 35     else if(amatch("goto", 4))    (dogoto(); laststst=STGOTO;)
 36     else if(dolabel())            ;
 37 #endif
 38     else if(amatch("return",6))   (doreturn();ns();laststst=STRETURN;)
 39     else if(amatch("break",5))    (dobreak();ns();laststst=STBREAK;)
 40     else if(amatch("continue",8))(docont();ns();laststst=STCONT;)
 41     else if(match(";"))           errflag=0;
 42     else if(match("#asm"))        (doasm();laststst=STASM;)
 43     else                          (doexpr();ns();laststst=STEXPR;)
 44     }
 45   return laststst;
 46   )
 47
```

```
113     expression(&const, &val);
114     clearstage(before, start);
115     if(ch != ',') break;
116     bump(1);
117     }
118   )
119
120 dowhile()  {
121   int wq[4];              /* allocate local queue */
122   addwhile(wq);           /* add entry to queue for "break" */
123   postlabel(wq[WQLOOP]);  /* loop label */
124   test(wq[WQEXIT], YES);  /* see if true */
125   statement();            /* if so, do a statement */
126   jump(wq[WQLOOP]);       /* loop to label */
127   postlabel(wq[WQEXIT]);  /* exit label */
128   delwhile();             /* delete queue entry */
129   }
130
131 #ifdef STDO
132 dodo() {
133   int wq[4], top;
134   addwhile(wq);
135   postlabel(top=getlabel());
136   statement();
137   needtoken("while");
138   postlabel(wq[WQLOOP]);
139   test(wq[WQEXIT], YES);
140   jump(top);
141   postlabel(wq[WQEXIT]);
142   delwhile();
143   ns();
144   }
145 #endif
146
147 #ifdef STFOR
148 dofor() {
149   int wq[4], lab1, lab2;
150   addwhile(wq);
151   lab1=getlabel();
152   lab2=getlabel();
153   needtoken("(");
154   if(match(";")==0) {
155     doexpr();             /* expr 1 */
156     ns();
157     }
158   postlabel(lab1);
159   if(match(";")==0) {
160     test(wq[WQEXIT], NO); /* expr 2 */
161     ns();
162     }
163   jump(lab2);
164   postlabel(wq[WQLOOP]);
165   if(match(")")==0) {
166     doexpr();             /* expr 3 */
167     needtoken(")");
168     }
169   jump(lab1);
170   postlabel(lab2);
171   statement();
172   jump(wq[WQLOOP]);
173   postlabel(wq[WQEXIT]);
174   delwhile();

175   }
176 #endif
177
```

```
/*
** semicolon enforcer
**
** called whenever syntax requires a semicolon
*/
ns()  {
  if(match(";")==0) error("no semicolon");
  else errflag=0;
  }

compound()  {

  int savcsp;
  char *savloc;
  savcsp=csp;
  savloc=locptr;
  declared=0;    /* may now declare local variables */
  ++ncmp;        /* new level open */
  while (match("}")==0)
    if(eof) {
      error("no final }");
      break;
      }
    else statement();     /* do one */
  --ncmp;                 /* close current level */
  csp=modstk(savcsp, NO); /* delete local variable space */
#ifdef STGOTO
  cptr=savloc;            /* retain labels */
  while(cptr < locptr) {
    cptr2=nextsym(cptr);
    if(cptr[IDENT] == LABEL) {
      while(cptr < cptr2) *savloc++ = *cptr++;
      }
    else cptr=cptr2;
    }
#endif
  locptr=savloc;          /* delete local symbols */
  declared = -1;          /* may not declare variables */
  }

doif()  {
  int flab1,flab2;
  flab1=getlabel(); /* get label for false branch */
  test(flab1, YES); /* get expression, and branch false */
  statement();      /* if true, do a statement */
  if (amatch("else",4)==0) {      /* if...else ? */
    /* simple "if"...print false label */
    postlabel(flab1);
    return;         /* and exit */
    }
  flab2=getlabel();
#ifdef STGOTO
  if((laststst != STRETURN)&(laststst != STGOTO)) jump(flab2);
#else
  if(laststst != STRETURN) jump(flab2);
#endif
  postlabel(flab1); /* print false label */
  statement();      /* and do "else" clause */
  postlabel(flab2); /* print true label */
  }

doexpr() {
  int const, val;
  char *before, *start;
  while(1) {
    setstage(&before, &start);
```

(Continued on top of page 38)

```
#ifdef STSWITCH
doswitch() {
  int wq[4], endlab, swact, swdef, *swnex, *swptr;
  swact=swactive;
  swdef=swdefault;
  swnex=swptr=swnext;
  addwhile(wq);
  needtoken("(");
  doexpr();       /* evaluate switch expression */
  needtoken(")");
  swdefault=0;
  swactive=1;
  jump(endlab=getlabel());
  statement();    /* cases, etc. */
  jump(wq[WQEXIT]);
  postlabel(endlab);
  sw();           /* match cases */
  while(swptr < swnext) {
    defstorage(CINT>>2);
    printlabel(*swptr++);  /* case label */
    outbyte(',');
    outdec(*swptr++);      /* case value */
    nl();
    }
  defstorage(CINT>>2);
  outdec(0);
  nl();
  if(swdefault) jump(swdefault);
  postlabel(wq[WQEXIT]);
  delwhile();
  swnext=swnex;
  swdefault=swdef;
  swactive=swact;
  }

docase() {
  if(swactive==0) error("not in switch");
  if(swnext > swend) {
    error("too many cases");
    return;
    }
  postlabel(*swnext++ = getlabel());
  constexpr(swnext++);
  needtoken(":");
  }

dodefault() {
  if(swactive) {
    if(swdefault) error("multiple defaults");
    }
  else error("not in switch");
  needtoken(":");
  postlabel(swdefault=getlabel());
  }
#endif

#ifdef STGOTO
dogoto() {
  if(nogo > 0) error("not allowed with block-locals");
  else noloc = 1;
  if(symname(ssname, YES)) jump(addlabel());
  else error("bad label");
  ns();
  }
```

(Continued on column 1, page 42)

Small-C Compiler, v.2

(Listing continued, text begins on page 16)

Listing One (Part two continued)

```
243 dolabel() {
244   char *savelptr;
245   blanks();
246   savelptr=lptr;
247   if(symname(ssname, YES)) {
248     if(gch()==':') {
249       postlabel(addlabel());
250       return 1;
251       }
252     else bump(savelptr-lptr);
253     }
254   return 0;
255   }
256
257 addlabel()  {
258   if(cptr=findloc(ssname)) {
259     if(cptr[IDENT]!=LABEL) error("not a label");
260     }
261   else cptr=addsym(ssname, LABEL, LABEL, getlabel(), &locptr, LABEL);
262   return (getint(cptr+OFFSET, OFFSIZE));
263   }
264 #endif
265
266 doreturn()  {
267   if(endst()==0) {
268     doexpr();
269     modstk(0, YES);
270     }
271   else modstk(0, NO);
272   ret();
273   }
274
275 dobreak()  {
276   int *ptr;
277   if ((ptr=readwhile())==0) return; /* no loops open */
278   modstk((ptr[WQSP]), NO);          /* clean up stk ptr */
279   jump(ptr[WQEXIT]);                /* jump to exit label */
280   }
281
282 docont()  {
283   int *ptr;
284   if ((ptr=readwhile())==0) return; /* no loops open */
285   modstk((ptr[WQSP]), NO);          /* clean up stk ptr */
286   jump(ptr[WQLOOP]);                /* jump to loop label */
287   }
288
289 doasm()  {
290   ccode=0;                  /* mark mode as "asm" */


291   while (1) {
292     inline();
293     if (match("#endasm")) break;
294     if(eof)break;
295     lout(line, output);
296     }
297   kill();
298   ccode=1;
299   }
300
```

```
61 #include cc21.c,2
62 #include cc22.c,2
63

 1 junk() {
 2   if(an(inbyte())) while(an(ch)) gch();
 3   else while(an(ch)==0) {
 4     if(ch==0) break;
 5     gch();
 6     }
 7   blanks();
 8   }
 9
10 endst() {
11   blanks();
12   return ((streq(lptr,";")|(ch==0)));
13   }
14
15 illname() {
16   error("illegal symbol");
17   junk();
18   }
19
20
21 multidef(sname)  char *sname; {
22   error("already defined");
23   }
24
25 needtoken(str)  char *str; {
26   if (match(str)==0) error("missing token");
27   }
28
29 needlval() {
30   error("must be lvalue");
31   }
32
33 findglb(sname)  char *sname; {
34 #ifdef HASH
35   if(search(sname, STARTGLB, SYMMAX, ENDGLB, NUMGLBS, NAME))
36     return cptr;
37 #else
38   cptr=STARTGLB;
39   while(cptr < glbptr) {
40     if(astreq(sname, cptr+NAME, NAMEMAX)) return cptr;
41     cptr=nextsym(cptr);
42     }
43 #endif
44   return 0;
45   }
46
47 findloc(sname)  char *sname;  {
48   cptr = locptr - 1;  /* search backward for block locals */
49   while(cptr > STARTLOC) {
50     cptr = cptr - *cptr;
51     if(astreq(sname, cptr, NAMEMAX)) return (cptr - NAME);
52     cptr = cptr - NAME - 1;
53     }
54   return 0;
55   }
56
57 addsym(sname, id, typ, value, lgptrptr, class)
58   char *sname, id, typ;  int value, *lgptrptr, class; {
59   if(lgptrptr == &glbptr) {
60     if(cptr2=findglb(sname)) return cptr2;
61 #ifdef HASH
62     if(cptr==0) {
63       error("global symbol table overflow");
64       return 0;
65       }
```

(Part three)

File: cc2.c,2

```
 1 /*
 2 ** Small-C Compiler Version 2.0
 3 **
 4 ** Copyright 1982 J. E. Hendrix
 5 **
 6 ** Part 2
 7 */
 8 #include stdio.h,2
 9 #include cc.def,2
10
11 /*
12 ** external references in part 1
13 */
14 extern char
15 #ifdef DYNAMIC
16  *symtab,
17  *stage,
18 #ifdef HASH
19  *macn,
20 #endif
21  *macq,
22  *pline,
23  *mline,
24 #else
25   symtab[SYMTBSZ],
26   stage[STAGESIZE],
27 #ifdef HASH
28   macn[MACNSIZE],
29 #endif
30   macq[MACQSIZE],
31   pline[LINESIZE],
32   mline[LINESIZE],
33 #endif
34   alarm, *glbptr, *line, *lptr, *cptr, *cptr2,  *cptr3,
35  *locptr, msname[NAMESIZE],  optimize,  pause,  quote[2],
36  *stagelast, *stagenext;
37 extern int
38 #ifdef DYNAMIC
39   *wq,
40 #else
41   wq[WQTABSZ],
42 #endif
43 #ifndef HASH
44   mack,
45 #endif
46   ccode,  ch,  csp,  eof,  errflag,  iflevel,
47   input,  input2,  listfp,  macptr,  nch,
48   nxtlab,  op[16],  opindex,  opsize,  output,  pptr,
49   skiplevel,  *wqptr;
50 extern int
51   openin();
52
53 /*
54 ** external references in part 4
55 */
56 #ifdef OPTIMIZE
57 extern int
58   peephole();
59 #endif
60
```

(Continued on top of page 42)

```
66 #else
67 #ifndef DYNAMIC
68     if(glbptr >= ENDGLB) {
69       error("global symbol table overflow");
70       return 0;
71       }
72 #endif
73     cptr=*lgptrptr;
74 #endif
75     }
76   else {
77     if(locptr > (ENDLOC-SYMMAX)) {
78       error("local symbol table overflow");
79       abort(ERRCODE);
80       }
81     cptr=*lgptrptr;
82     }
83   cptr[IDENT]=id;
84   cptr[TYPE]=typ;
85   cptr[CLASS]=class;
86   putint(value, cptr+OFFSET, OFFSIZE);
87   cptr3 = cptr2 = cptr + NAME;
88   while(an(*sname)) *cptr2++ = *sname++;
89 #ifdef HASH
90   if(lgptrptr == &locptr) {
91     *cptr2 = cptr2 - cptr3;          /* set length */
92     *lgptrptr = ++cptr2;
93     }
94 #else
95   *cptr2 = cptr2 - cptr3;          /* set length */
96   *lgptrptr = ++cptr2;
97 #ifdef DYNAMIC
98   if(lgptrptr == &glbptr) CCALLOC(cptr2 - cptr);
99   /*  gets allocation error if no more memory  */
100 #endif
101 #endif
102   return cptr;
103   }
104
105 #ifndef HASH
106 nextsym(entry) char *entry; {
107   entry = entry + NAME;
108   while(*entry++ >= ' '); /* find length byte */
109   return entry;
110   }
111 #endif
112
113 /*
114 ** get integer of length len from address addr
115 ** (byte sequence set by "putint")
116 */
117 getint(addr, len) char *addr; int len; {
118   int i;
119   i = *(addr + --len);  /* high order byte sign extended */
120   while(len--) i = (i << 8) | *(addr+len)&255;
121   return i;
122   }
123
124 /*
125 ** put integer i of length len into address addr
126 ** (low byte first)
127 */
128 putint(i, addr, len) char *addr; int i, len; {
129   while(len--) {
130     *addr++ = i;
```

(Continued on next page)

Small-C Compiler, v.2

(Listing continued, text begins on page 16)

Listing One (Part three continued)

```
131       i = i>>8;
132       }
133     }
134
135 /*
136 ** test if next input string is legal symbol name
137 */
138 symname(sname, ucase) char *sname; int ucase; {
139     int k;char c;
140     blanks();
141     if(alpha(ch)==0) return 0;
142     k=0;
143     while(an(ch)) {
144 #ifdef UPPER
145       if(ucase)
146         sname[k]=upper(gch());
147       else
148 #endif
149         sname[k]=gch();
150       if(k<NAMEMAX) ++k;
151       }
152     sname[k]=0;
153     return 1;
154     }
155
156 #ifdef UPPER
157 /*
158 ** force upper case alphabetics
159 */
160 upper(c)  char c; {
161     if((c >= 'a') & (c <= 'z')) return (c - 32);
162     else return c;
163     }
164 #endif
165
166 /*
167 ** return next avail internal label number
168 */
169 getlabel() {
170     return(++nxtlab);
171     }
172
173 /*
174 ** post a label in the program
175 */
176 postlabel(label) int label; {
177     printlabel(label);
178     col();
179     nl();
180     }
181
182 /*
183 ** print specified number as a label
184 */
185 printlabel(label)  int label; {
186     outstr("CC");
187     outdec(label);
188     }
189
```

```
255     else  lptr=line;
256     if(ch=nch=*lptr) nch=*(lptr+1);
257     }
258
259 kill() {
260     *line=0;
261     bump(0);
262     }
263
264 inbyte()  {
265     while(ch==0) {
266       if (eof) return 0;
267       preprocess();
268       }
269     return gch();
270     }
271
272 inline() {
273     int k,unit;
274 #ifdef POLL
275     CCPOLL(1);  /* allow program interruption */
276 #endif
277     while(1) {
278       if (input==EOF) openin();
279       if(eof) return;
280       if((unit=input2)==EOF) unit=input;
281       if(fgets(line, LINEMAX, unit)==NULL) {
282         fclose(unit);
283         if(input2!=EOF) input2=EOF;
284         else input=EOF;
285         }
286       else {
287         bump(0);
288         return;
289         }
290       }
291     }
292


  1 ifline() {
  2     while(1) {
  3       inline();
  4       if(eof) return;
  5       if(match("#ifdef")) {
  6         ++iflevel;
  7         if(skiplevel) continue;
  8         blanks();
  9 #ifdef HASH
 10         if(search(lptr, macn, NAMESIZE+2, MACNEND, MACNBR, 0)==0)
 11 #else
 12         if(findmac(lptr)==0)
 13 #endif
 14           skiplevel=iflevel;
 15         continue;
 16         }
 17       if(match("#ifndef")) {
 18         ++iflevel;
 19         if(skiplevel) continue;
 20         blanks();
 21 #ifdef HASH
 22         if(search(lptr, macn, NAMESIZE+2, MACNEND, MACNBR, 0))
 23 #else
 24         if(findmac(lptr))
 25 #endif
 26           skiplevel=iflevel;
 27         continue;
 28         }
 29       if(match("#else")) {
```

```
190 /*
191 ** test if given character is alphabetic
192 */
193 alpha(c)  char c; {
194   return (((c>='a')&(c<='z'))|((c>='A')&(c<='Z'))|(c=='_'));
195   }
196
197 /*
198 ** test if given character is numeric
199 */
200 numeric(c)  char c; {
201   return((c>='0')&(c<='9'));
202   }
203
204 /*
205 ** test if given character is alphanumeric
206 */
207 an(c)  char c; {
208   return ((alpha(c))|(numeric(c)));
209   }
210
211 addwhile(ptr)  int ptr[]; {
212   int k;
213   ptr[WQSP]=csp;           /* and stk ptr */
214   ptr[WQLOOP]=getlabel();  /* and looping label */
215   ptr[WQEXIT]=getlabel();   /* and exit label */
216   if (wqptr==WQMAX) {
217     error("too many active loops");
218     abort(ERRCODE);
219     }
220   k=0;
221   while (k<WQSIZ) *wqptr++ = ptr[k++];
222   }
223
224 delwhile() {
225   if(readwhile()) wqptr=wqptr-WQSIZ;
226   }
227
228 readwhile() {
229   if (wqptr==wq) {
230     error("no active loops");
231     return 0;
232     }
233   else return (wqptr-WQSIZ);
234  }
235
236 white() {
237   /* test for stack/program overlap */
238   /* primary -> symname -> blanks -> white */
239 #ifdef DYNAMIC
240   CCAVAIL();  /* abort on stack/symbol table overflow */
241 #endif
242   if(*lptr==' ') return 1;
243   if(*lptr==9)   return 1;
244   return 0;
245   }
246
247 gch() {
248   int c;
249   if(c=ch) bump(1);
250   return c;
251   }
252
253 bump(n) int n; {
254   if(n) lptr=lptr+n;
```

(Continued on top of page 44)

```
30      if(iflevel) {
31        if(skiplevel==iflevel) skiplevel=0;
32        else if(skiplevel==0)  skiplevel=iflevel;
33        }
34      else noiferr();
35      continue;
36      }
37    if(match("#endif")) {
38      if(iflevel) {
39        if(skiplevel==iflevel) skiplevel=0;
40        --iflevel;
41        }
42      else noiferr();
43      continue;
44      }
45    if(skiplevel) continue;
46    if(listfp) {
47      if(listfp==output) cout('#', output);
48      lout(line, listfp);
49      }
50    if(ch==0) continue;
51    break;
52    }
53  }
54
55 keepch(c)  char c; {
56   if(pptr<LINEMAX) pline[++pptr]=c;
57   }
58

59 preprocess() {
60   int k;
61   char c;
62   if(ccode) {
63     line=mline;
64     ifline();
65     if(eof) return;
66     }
67   else {
68     line=pline;
69     inline();
70     return;
71     }
72   pptr = -1;
73   while(ch) {
74     if(white()) {
75       keepch(' ');
76       while(white()) gch();
77       }
78     else if(ch=='"') {
79       keepch(ch);
80       gch();
81       while((ch!='"')|((*(lptr-1)==92)&(*(lptr-2)!=92))) {
82         if(ch==0) {
83           error("no quote");
84           break;
85           }
86         keepch(gch());
87         }
88       gch();
89       keepch('"');
90       }
91     else if(ch==39) {
92       keepch(39);
93       gch();
94       while((ch!=39)|((*(lptr-1)==92)&(*(lptr-2)!=92))) {
```

(Continued on next page)

Small-C Compiler, v.2

(Listing continued, text begins on page 16)

Listing One (Part three continued)

```
95          if(ch==0) {
96            error("no apostrophe");
97            break;
98            }
99          keepch(gch());
100         }
101       gch();
102       keepch(39);
103       }
104     else if((ch=='/')&(nch=='*')) {
105       bump(2);
106       while(((ch=='*')&(nch=='/'))==0) {
107         if(ch) bump(1);
108         else {
109           ifline();
110           if(eof) break;
111           }
112         }
113       bump(2);
114       }
115     else if(an(ch)) {
116       k=0;

117       while(an(ch)) {
118         if(k<NAMEMAX) msname[k++]=ch;
119         gch();
120         }
121       msname[k]=0;
122 #ifdef HASH
123       if(search(msname, macn, NAMESIZE+2, MACNEND, MACNBR, 0)) {
124         k=getint(cptr+NAMESIZE, 2);
125         while(c=macq[k++]) keepch(c);
126         }
127 #else
128       if(k=findmac(msname)) while(c=macq[k++]) keepch(c);
129 #endif
130       else {
131         k=0;
132         while(c=msname[k++]) keepch(c);
133         }
134       }
135     else keepch(gch());
136     }
137   if(pptr>=LINEMAX) error("line too long");
138   keepch(0);
139   line=pline;
140   bump(0);
141   }
142
143 noiferr() {
144   error("no matching #if...");
145   errflag=0;
146   }
147
148 addmac() {
149   int k;
150   if(symname(msname, NO)==0) {
151     illname();
152     kill();
153     return;
```

```
219
220 setstage(before, start) int *before, *start; {
221   if((*before=stagenext)==0) stagenext=stage;
222   *start=stagenext;
223   }
224
225 clearstage(before, start) char *before, *start; {
226   *stagenext=0;
227   if(stagenext=before) return;
228   if(start) {
229 #ifdef OPTIMIZE
230     peephole(start);
231 #else
232     sout(start, output);

233 #endif
234     }
235   }
236
237 outdec(number)  int number; {
238   int k,zs;
239   char c;
240   zs = 0;
241   k=10000;
242   if (number<0) {
243     number=(-number);
244     outbyte('-');
245     }
246   while (k>=1) {
247     c=number/k + '0';
248     if ((c!='0')|(k==1)|(zs)) {
249       zs=1;
250       outbyte(c);
251       }
252     number=number%k;
253     k=k/10;
254     }
255   }
256
257 ol(ptr)  char ptr[];  {
258   ot(ptr);
259   nl();
260   }
261
262 ot(ptr) char ptr[]; {
263 #ifdef TAB
264   tab();
265 #endif
266   outstr(ptr);
267   }
268
269 outstr(ptr) char ptr[]; {
270 #ifdef POLL
271   CCPOLL(1); /* allow program interruption */
272 #endif
273   /* must work with symbol table names terminated by length */
274   while(*ptr >= ' ') outbyte(*ptr++);
275   }
276
277 outbyte(c) char c; {
278   if(stagenext) {
279     if(stagenext==stagelast) {
280       error("staging buffer overflow");
281       return 0;
282       }
283     else *stagenext++ = c;
284     }
285   else cout(c,output);
286   return c;
```

```
154     }
155   k=0;
156 #ifdef HASH
157   if(search(msname, macn, NAMESIZE+2, MACNEND, MACNBR, 0)==0) {
158     if(cptr2=cptr) while(*cptr2++ = msname[k++]);
159     else {
160       error("macro name table full");
161       return;
162       }
163     }
164   putint(macptr, cptr+NAMESIZE, 2);
165 #else
166   while(putmac(msname[k++]));
167 #endif
168   while(white()) gch();
169   while(putmac(gch()));
170   if(macptr>=MACMAX) {
171     error("macro string queue full"); abort(ERRCODE);
172     }
173   }
174
175 putmac(c)  char c; {
176   macq[macptr]=c;
177   if(macptr<MACMAX) ++macptr;
178   return c;
179   }
180
181 #ifdef HASH
182 /*
183 ** search for symbol match
184 ** on return cptr points to slot found or empty slot
185 */
186 search(sname, buf, len, end, max, off)
187   char *sname, *buf, *end;  int len, max, off; {
188   cptr=cptr2=buf+((hash(sname)%(max-1))*len);
189   while(*cptr != 0) {
190     if(astreq(sname, cptr+off, NAMEMAX)) return 1;
191     if((cptr=cptr+len) >= end) cptr=buf;
192     if(cptr == cptr2) return (cptr=0);
193     }
194   return 0;
195   }
196
197 hash(sname) char *sname; {
198   int i, c;
199   i=0;
200   while(c=*sname++) i=(i<<1)+c;
201   return i;
202   }
203
204 #else
205
206 findmac(sname)  char *sname; {
207   mack=0;
208   while(mack<macptr) {
209     if(astreq(sname,macq+mack,NAMEMAX)) {
210       while(macq[mack++]);
211       return mack;
212       }
213     while(macq[mack++]);
214     while(macq[mack++]);
215     }
216   return 0;
217   }
218 #endif
```

(Continued on top of page 46)

```
287   }
288
289 cout(c, fd) char c; int fd; {
290   if(fputc(c, fd)==EOF) xout();
291   }
292
293 sout(string, fd) char *string; int fd; {
294   if(fputs(string, fd)==EOF) xout();
295   }
296
297 lout(line, fd) char *line; int fd; {
298   sout(line, fd);
299   cout('\n', fd);
300   }
301
302 xout() {
303   fputs("output error\n", stderr);
304   abort(ERRCODE);
305   }
306
307 nl() {
308   outbyte('\n');
309   }
310
311 tab() {
312   outbyte(TAB);
313   }
314
315 col() {
316 #ifdef COL
317   outbyte(':');
318 #endif
319   }
320
321 error(msg) char msg[]; {
322   if(errflag) return; else errflag=1;
323   lout(line, stderr);
324   errout(msg, stderr);
325   if(alarm) fputc(7, stderr);
326   if(pause) while(fgetc(stderr)!='\n');
327   if(listfp>0) errout(msg, listfp);
328   }
329
330 errout(msg, fp) char msg[]; int fp; {
331   int k; k=line+2;
332   while(k++ <= lptr) cout(' ', fp);
333   lout("/\\", fp);
334   sout("**** ", fp); lout(msg, fp);
335   }
336
337 streq(str1,str2)  char str1[],str2[]; {
338   int k;
339   k=0;
340   while (str2[k]) {
341     if ((str1[k])!=(str2[k])) return 0;
342     ++k;
343     }
344   return k;
345 }
346
347 astreq(str1,str2,len)  char str1[],str2[];int len; {
348   int k;

349   k=0;
350   while (k<len) {
```

(Continued on next page)

Small-C Compiler, v.2

(Listing continued, text begins on page 16)

Listing One (Part three continued)

```
351     if ((str1[k])!=(str2[k]))break;
352     /*
353     ** must detect end of symbol table names terminated by
354     ** symbol length in binary
355     */
356     if(str1[k] < ' ') break;
357     if(str2[k] < ' ') break;
358     ++k;
359     )
360   if (an(str1[k]))return 0;
361   if (an(str2[k]))return 0;
362   return k;
363  )
364
365 match(lit)  char *lit; {
366   int k;
367   blanks();
368   if (k=streq(lptr,lit)) {
369     bump(k);
370     return 1;
371     }
372   return 0;
373   }
374
375 amatch(lit,len)  char *lit;int len; {
376   int k;
377   blanks();
378   if (k=astreq(lptr,lit,len)) {
379     bump(k);
380     while(an(ch)) inbyte();
381     return 1;
382     }
383   return 0;
384  }
385
386 nextop(list) char *list; {
387   char op[4];
388   opindex=0;
389   blanks();
390   while(1) {
391     opsize=0;
392     while(*list > ' ') op[opsize++]=*list++;
393     op[opsize]=0;
394     if(opsize=streq(lptr, op))
395       if((*(lptr+opsize) != '=')&
396          (*(lptr+opsize) != *(lptr+opsize-1)))
397         return 1;
398     if(*list) {
399       ++list;
400       ++opindex;
401       }
402     else return 0;
403     }
404   }
405
406 blanks() {

407   while(1) {
408     while(ch) {
409       if(white()) gch();
```

```
55 #include cc33.c,2
56
 1 /*
 2 ** lval[0] - symbol table address, else 0 for constant
 3 ** lval[1] - type of indirect obj to fetch, else 0 for static
 4 ** lval[2] - type of pointer or array, else 0 for all other
 5 ** lval[3] - true if constant expression
 6 ** lval[4] - value of constant expression
 7 ** lval[5] - true if secondary register altered
 8 ** lval[6] - function address of highest/last binary operator
 9 ** lval[7] - stage address of "oper 0" code, else 0
10 */
11
12 /*
13 ** skim over terms adjoining || and && operators
14 */
15 skim(opstr, testfunc, dropval, endval, heir, lval)
16   char *opstr;
17   int testfunc, dropval, endval, heir, lval[]; {
18   int k, hits, droplab, endlab;
19   hits=0;
20   while(1) {
21     k=plunge1(heir, lval);
22     if(nextop(opstr)) {
23       bump(opsize);
24       if(hits==0) {
25         hits=1;
26         droplab=getlabel();
27         }
28       dropout(k, testfunc, droplab, lval);
29       }
30     else if(hits) {
31       dropout(k, testfunc, droplab, lval);
32       const(endval);
33       jump(endlab=getlabel());
34       postlabel(droplab);
35       const(dropval);
36       postlabel(endlab);
37       lval[1]=lval[2]=lval[3]=lval[7]=0;
38       return 0;
39       }
40     else return k;
41     }
42   }
43
44 /*
45 ** test for early dropout from || or && evaluations
46 */
47 dropout(k, testfunc, exitl, lval) int k, testfunc, exitl, lval[]; {
48   if(k) rvalue(lval);
49   else if(lval[3]) const(lval[4]);
50   testfunc(exitl); /* jumps on false */
51   }
52
53 /*
54 ** plunge to a lower level
55 */
56 plunge(opstr, opoff, heir, lval)
57   char *opstr;
58   int opoff, heir, lval[]; {

59   int k, lval2[8];
60   k=plunge1(heir, lval);
61   if(nextop(opstr)==0) return k;
62   if(k) rvalue(lval);
63   while(1) {
64     if(nextop(opstr)) {
65       bump(opsize);
66       opindex=opindex+opoff;
```

```
410       else return;
411       )
412     if(line==mline) return;
413     preprocess();
414     if(eof)break;
415     )
416   )
417
```

(Part four)

```
  1 /*
  2 ** Small-C Compiler Version 2.0
  3 **
  4 ** Copyright 1982 J. E. Hendrix
  5 **
  6 ** Part 3
  7 */
  8 #include stdio.h,2
  9 #include cc.def,2
 10
 11 /*
 12 ** external references in part 1
 13 */
 14 extern char
 15 #ifdef DYNAMIC
 16  *stage,
 17  *litq,
 18 #else
 19   stage[STAGESIZE],
 20   litq[LITABSZ],
 21 #endif
 22  *glbptr, *lptr,  ssname[NAMESIZE],  quote[2], *stagenext;
 23 extern int
 24   ch,  csp,  litlab,  litptr,  nch,  op[16],  op2[16],
 25   oper,  opindex,  opsize;
 26
 27 /*
 28 ** external references in part 2
 29 */
 30 extern int
 31   addsym(),  blanks(),  bump(),  clearstage(),  endst(),
 32   error(),  findglb(),  findloc(),  gch(),  getlabel(),
 33   inbyte(),  junk(),  match(),  needlval(),  needtoken(),
 34   nextop(),  nl(),  numeric(),  outbyte(),  outdec(),
 35   outstr(),  postlabel(),  printlabel(),  putint(),
 36   setstage(),  streq(),  symname();
 37
 38 /*
 39 ** external references in part 4
 40 */
 41 extern int
 42   add(),  and(),  asl(),  asr(),  call(),  callstk(),
 43   com(),  dec(),  div(),  doublereg(),  eq(),  eq0(),
 44   ge(),  ge0(),  getloc(),  getmem(),  gt(),  gt0(),
 45   immed(),  immed2(),  inc(),  indirect(),  jump(),
 46   le(),  le0(),  lneg(),  loadargc(),  lt(),  lt0(),
 47   mod(),  modstk(),  move(),  mult(),  ne(),  ne0(),
 48   neg(),  or(),  pop(),  push(),  putmem(),
 49   putstk(),  ret(),  smartpop(),  sub(),  swap(),  swapstk(),
 50   testjump(),  uge(),  ugt(),  ule(),  ult(),  ult0(),
 51   xor(),  zerojump();
 52
 53 #include cc31.c,2
 54 #include cc32.c,2
```

(Continued on top of page 48)

```
 67       plunge2(op[opindex], op2[opindex], heir, lval, lval2);
 68       }
 69     else return 0;
 70     }
 71   }
 72
 73 /*
 74 ** unary plunge to lower level
 75 */
 76 plunge1(heir, lval) int heir, lval[]; {
 77   char *before, *start;
 78   int k;
 79   setstage(&before, &start);
 80   k=heir(lval);
 81   if(lval[3]) clearstage(before,0); /* load constant later */
 82   return k;
 83   }
 84
 85 /*
 86 ** binary plunge to lower level
 87 */
 88 plunge2(oper, oper2, heir, lval, lval2)
 89   int oper, oper2, heir, lval[], lval2[]; {
 90   char *before, *start;
 91   setstage(&before, &start);
 92   lval[5]=1;          /* flag secondary register used */
 93   lval[7]=0;          /* flag as not "... oper 0" syntax */
 94   if(lval[3]) {       /* constant on left side not yet loaded */
 95     if(plunge1(heir, lval2)) rvalue(lval2);
 96     if(lval[4]==0) lval[7]=stagenext;
 97     const2(lval[4]<<dbltest(lval2, lval));
 98     }
 99   else {              /* non-constant on left side */
100     push();
101     if(plunge1(heir, lval2)) rvalue(lval2);
102     if(lval2[3]) {    /* constant on right side */
103       if(lval2[4]==0) lval[7]=start;
104       if(oper==add) { /* may test other commutative operators */
105         csp=csp+2;
106         clearstage(before, 0);
107         const2(lval2[4]<<dbltest(lval, lval2));   /* load secondary */
108         }
109       else {
110         const(lval2[4]<<dbltest(lval, lval2));    /* load primary */
111         smartpop(lval2, start);
112         }
113       }
114     else {            /* non-constants on both sides */
115       smartpop(lval2, start);
116       if((oper==add)|(oper==sub)) {
117         if(dbltest(lval,lval2)) doublereg();
118         if(dbltest(lval2,lval)) {
119           swap();
120           doublereg();
121           if(oper==sub) swap();
122           }
123         }
124       }
125     }
126   if(oper) {
127     if(lval[3]=lval[3]&lval2[3]) {
128       lval[4]=calc(lval[4], oper, lval2[4]);
129       clearstage(before, 0);
130       lval[5]=0;
131       }
```

(Continued on next page)

Small-C Compiler, v.2

(Listing continued, text begins on page 16)

Listing One (Part four continued)

```
132     else {
133       if((lval[2]==0)&(lval2[2]==0)) {
134         oper();
135         lval[6]=oper;     /* identify the operator */
136         }
137       else {
138         oper2();
139         lval[6]=oper2;    /* identify the operator */
140         }
141       }
142     if(oper==sub) {
143       if((lval[2]==CINT)&(lval2[2]==CINT)) {
144         swap();
145         const(1);
146         asr();  /** div by 2 **/
147         }
148       }
149     if((oper==sub)|(oper==add)) result(lval, lval2);
150     }
151   }
152
153 calc(left, oper, right) int left, oper, right; {
154         if(oper ==  or) return (left  |  right);
155   else if(oper == xor) return (left  ^  right);
156   else if(oper == and) return (left  &  right);
157   else if(oper ==  eq) return (left  == right);
158   else if(oper ==  ne) return (left  != right);
159   else if(oper ==  le) return (left  <= right);
160   else if(oper ==  ge) return (left  >= right);
161   else if(oper ==  lt) return (left  <  right);
162   else if(oper ==  gt) return (left  >  right);
163   else if(oper == asr) return (left  >> right);
164   else if(oper == asl) return (left  << right);
165   else if(oper == add) return (left  +  right);
166   else if(oper == sub) return (left  -  right);
167   else if(oper ==mult) return (left  *  right);
168   else if(oper == div) return (left  /  right);
169   else if(oper == mod) return (left  %  right);
170   else return 0;
171   }
172
173 expression(const, val) int *const, *val;  {
174   int lval[8];
175   if(heir1(lval)) rvalue(lval);
176   if(lval[3]) {
177     *const=1;
178     *val=lval[4];
179     }
180   else *const=0;
181   }
182
183 heir1(lval)  int lval[];  {
184   int k,lval2[8], oper;
185   k=plunge1(heir3, lval);
186   if(lval[3]) const(lval[4]);
187         if(match("|="))  oper=or;
188   else if(match("^="))  oper=xor;
189   else if(match("&="))  oper=and;
190   else if(match("+="))  oper=add;
191   else if(match("-="))  oper=sub;
```

```
257 heir11(lval)  int lval[]; {
258   return plunge("+ -", 11, heir12, lval);
259   }
260
261 heir12(lval)  int lval[]; {
262   return plunge("* / %", 13, heir13, lval);
263   }
264
  1 heir13(lval)  int lval[]; {
  2   int k;
  3   char *ptr;
  4   if(match("++")) {                    /* ++lval */
  5     if(heir13(lval)==0) {
  6       needlval();
  7       return 0;
  8       }
  9     step(inc, lval);
 10     return 0;
 11     }
 12   else if(match("--")) {               /* --lval */
 13     if(heir13(lval)==0) {
 14       needlval();
 15       return 0;
 16       }
 17     step(dec, lval);
 18     return 0;
 19     }
 20   else if (match("~")) {               /* ~ */
 21     if(heir13(lval)) rvalue(lval);
 22     com();
 23 #ifdef PHASE2
 24     lval[4] = ~lval[4];
 25 #endif
 26     return 0;
 27     }
 28   else if (match("!")) {               /* ! */
 29     if(heir13(lval)) rvalue(lval);
 30     lneg();
 31 #ifdef PHASE2
 32     lval[4] = !lval[4];
 33 #endif
 34     return 0;
 35     }
 36   else if (match("-")) {               /* unary - */
 37     if(heir13(lval)) rvalue(lval);
 38     neg();
 39     lval[4] = -lval[4];
 40     return 0;
 41     }
 42   else if(match("*")) {                /* unary * */
 43     if(heir13(lval)) rvalue(lval);
 44     if(ptr=lval[0])lval[1]=ptr[TYPE];
 45     else lval[1]=CINT;
 46     lval[2]=0;  /* flag as not pointer or array */
 47     lval[3]=0;  /* flag as not constant */
 48     return 1;
 49     }
 50   else if(match("&")) {                /* unary & */
 51     if(heir13(lval)==0) {
 52       error("illegal address");
 53       return 0;
 54       }
 55     ptr=lval[0];
 56     lval[2]=ptr[TYPE];
 57     if(lval[1]) return 0;
 58     /* global & non-array */
 59     address(ptr);
 60     lval[1]=ptr[TYPE];
```

```
192    else if(match("*="))  oper=mult;
193    else if(match("/="))  oper=div;
194    else if(match("%="))  oper=mod;
195    else if(match(">>=")) oper=asr;
196    else if(match("<<=")) oper=asl;
197    else if(match("="))   oper=0;
198    else return k;
199    if(k==0) {
200      needlval();
201      return 0;
202      }
203    if(lval[1]) {
204      if(oper) {
205        push();
206        rvalue(lval);
207        }
208      plunge2(oper, oper, heir1, lval, lval2);
209      if(oper) pop();
210      }
211    else {
212      if(oper) {
213        rvalue(lval);
214        plunge2(oper, oper, heir1, lval, lval2);
215        }
216      else {
217        if(heir1(lval2)) rvalue(lval2);
218        lval[5]=lval2[5];
219        }
220      }
221    store(lval);
222    return 0;
223    }
224
225  heir3(lval)  int lval[]; {
226    return skim(";;", eq0, 1, 0, heir4, lval);
227    }
228
229  heir4(lval)  int lval[]; {
230    return skim("&&", ne0, 0, 1, heir5, lval);
231    }
232
233  heir5(lval)  int lval[]; {
234    return plunge("|", 0, heir6, lval);
235    }
236
237  heir6(lval)  int lval[]; {
238    return plunge("^", 1, heir7, lval);
239    }
240
241  heir7(lval)  int lval[]; {
242    return plunge("&", 2, heir8, lval);
243    }
244
245  heir8(lval)  int lval[];  {
246    return plunge("== !=", 3, heir9, lval);
247    }
248
249  heir9(lval)  int lval[];  {
250    return plunge("<= >= < >", 5, heir10, lval);
251    }
252
253  heir10(lval)  int lval[];  {
254    return plunge(">> <<", 9, heir11, lval);
255    }
256
```

(Continued on top of page 50)

```
61     return 0;
62     }
63   else {
64     k=heir14(lval);
65     if(match("++")) {                    /* lval++ */
66       if(k==0) {
67         needlval();
68         return 0;
69         }
70       step(inc, lval);
71       dec(lval[2]>>2);
72       return 0;
73       }
74     else if(match("--")) {               /* lval-- */
75       if(k==0) {
76         needlval();
77         return 0;
78         }
79       step(dec, lval);
80       inc(lval[2]>>2);
81       return 0;
82       }
83     else return k;
84     }
85   }
86
87  heir14(lval)  int *lval; {
88    int k, const, val, lval2[8];
89    char *ptr, *before, *start;
90    k=primary(lval);
91    ptr=lval[0];
92    blanks();
93    if((ch=='[')|(ch=='(')) {
94      lval[5]=1;     /* secondary register will be used */
95      while(1) {
96        if(match("[")) {                 /* [subscript] */
97          if(ptr==0) {
98            error("can't subscript");
99            junk();
100           needtoken("]");
101           return 0;
102           }
103         else if(ptr[IDENT]==POINTER)rvalue(lval);
104         else if(ptr[IDENT]!=ARRAY) {
105           error("can't subscript");
106           k=0;
107           }
108         setstage(&before, &start);
109         lval2[3]=0;
110         plunge2(0, 0, heir1, lval2, lval2); /* lval2 deadend */
111         needtoken("]");
112         if(lval2[3]) {
113           clearstage(before, 0);
114           if(lval2[4]) {
115             if(ptr[TYPE]==CINT) const2(lval2[4]<<LBPW);
116             else                const2(lval2[4]);
117             add();
118             }
119           }
120         else {
121           if(ptr[TYPE]==CINT) doublereg();
122           add();
123           }
124         lval[0]=lval[2]=0;
125         lval[1]=ptr[TYPE];
```

(Continued on next page)

Small-C Compiler, v.2

(Listing continued, text begins on page 16)

Listing One (Part four continued)

```
126         k=1;
127         }
128       else if(match("(")) {           /* function(...) */
129         if(ptr==0) callfunction(0);
130         else if(ptr[IDENT]!=FUNCTION) {
131           rvalue(lval);
132           callfunction(0);
133           }
134         else callfunction(ptr);
135         k=lval[0]=lval[3]=0;
136         }
137       else return k;
138       }
139     }
140   if(ptr==0) return k;
141   if(ptr[IDENT]==FUNCTION) {
142     address(ptr);
143     return 0;
144     }
145   return k;
146   }
147
148 primary(lval)  int *lval; {
149   char *ptr;
150   int k;
151   if(match("(")) {                    /* (expression) */
152     k=heir1(lval);
153     needtoken(")");
154     return k;
155     }
156   putint(0, lval, 8<<LBPW); /* clear lval array */
157   if(symname(ssname, YES)) {
158     if(ptr=findloc(ssname)) {
159 #ifdef STGOTO
160       if(ptr[IDENT]==LABEL) {
161         experr();
162         return 0;
163         }
164 #endif
165       getloc(ptr);
166       lval[0]=ptr;
167       lval[1]=ptr[TYPE];
168       if(ptr[IDENT]==POINTER) {
169         lval[1]=CINT;
170         lval[2]=ptr[TYPE];
171         }
172       if(ptr[IDENT]==ARRAY) {
173         lval[2]=ptr[TYPE];
174         return 0;

175         }
176       else return 1;
177       }
178     if(ptr=findglb(ssname))
179       if(ptr[IDENT]!=FUNCTION) {
180         lval[0]=ptr;
181         lval[1]=0;
182         if(ptr[IDENT]!=ARRAY) {
183           if(ptr[IDENT]==POINTER) lval[2]=ptr[TYPE];
184           return 1;
```

```
26      if(lval[5]) {
27        push();
28        rvalue(lval);
29        oper(lval[2]>>2);
30        pop();
31        store(lval);
32        return;
33        }
34      else {
35        move();
36        lval[5]=1;
37        }
38      }
39    rvalue(lval);
40    oper(lval[2]>>2);
41    store(lval);
42    }
43
44 store(lval)  int lval[]; {
45    if(lval[1]) putstk(lval);
46    else        putmem(lval);
47    }
48
49 rvalue(lval) int lval[]; {
50    if ((lval[0]!=0)&(lval[1]==0)) getmem(lval);
51    else                           indirect(lval);
52    }
53
54 test(label, parens)  int label, parens;  {
55    int lval[8];
56    char *before, *start;
57    if(parens) needtoken("(");
58    while(1) {
59      setstage(&before, &start);
60      if(heir1(lval)) rvalue(lval);
61      if(match(",")) clearstage(before, start);
62      else break;
63      }
64    if(parens) needtoken(")");
65    if(lval[3]) {  /* constant expression */
66      clearstage(before, 0);
67      if(lval[4]) return;
68      jump(label);
69      return;
70      }
71    if(lval[7]) {  /* stage address of "oper 0" code */
72      oper=lval[6];/* operator function address */
73           if((oper==eq);
74              (oper==ule)) zerojump(eq0, label, lval);
75      else if((oper==ne);
76              (oper==ugt)) zerojump(ne0, label, lval);
77      else if (oper==gt)   zerojump(gt0, label, lval);
78      else if (oper==ge)   zerojump(ge0, label, lval);
79      else if (oper==uge)  clearstage(lval[7],0);
80      else if (oper==lt)   zerojump(lt0, label, lval);
81      else if (oper==ult)  zerojump(ult0, label, lval);
82      else if (oper==le)   zerojump(le0, label, lval);
83      else                 testjump(label);
84      }
85    else testjump(label);
86    clearstage(before, start);
87    }
88
89 constexpr(val) int *val; {
90    int const;
91    char *before, *start;
92    setstage(&before, &start);
93    expression(&const, val);
94    clearstage(before, 0);  /* scratch generated code */
```

```
185           )
186         address(ptr);
187         lval[1]=lval[2]=ptr[TYPE];
188         return 0;
189         )
190     ptr=addsym(ssname, FUNCTION, CINT, 0, &glbptr, STATIC);
191     lval[0]=ptr;
192     lval[1]=0;
193     return 0;
194     )
195   if(constant(lval)==0) experr();
196   return 0;
197   )
198
199 experr() {
200   error("invalid expression");
201   const(0);
202   junk();
203   )
204
205 callfunction(ptr)  char *ptr; { /* symbol table entry or 0 */
206   int nargs, const, val;
207   nargs=0;
208   blanks();               /* already saw open paren */
209   if(ptr==0) push();      /* calling HL */
210   while(streq(lptr,")")==0) {
211     if(endst()) break;
212     expression(&const, &val);
213     if(ptr==0) swapstk(); /* don't push addr */
214     push();               /* push argument */
215     nargs=nargs+BPW;      /* count args*BPW */
216     if (match(",")==0) break;
217     }
218   needtoken(")");
219   if(streq(ptr+NAME, "CCARGC")==0) loadargc(nargs>>LBPW);
220   if(ptr) call(ptr+NAME);
221   else callstk();
222   csp=modstk(csp+nargs, YES);
223   )
224
```

```
  1 /*
  2 ** true if val1 -> int pointer or int array and val2 not ptr or array
  3 */
  4 dbltest(val1,val2) int val1[], val2[]; {
  5   if(val1[2]!=CINT) return 0;
  6   if(val2[2]) return 0;
  7   return 1;
  8   )
  9
 10 /*
 11 ** determine type of binary operation
 12 */
 13 result(lval, lval2) int lval1[], lval2[]; {
 14   if((lval[2]!=0)&(lval2[2]!=0)) {
 15     lval[2]=0;
 16     )
 17   else if(lval2[2]) {
 18     lval[0]=lval2[0];
 19     lval[1]=lval2[1];
 20     lval[2]=lval2[2];
 21     )
 22   )
 23
 24 step(oper, lval) int oper, lval[]; {
 25   if(lval[1]) {
```

(Continued on top of page 52)

```
 95   if(const==0) error("must be constant expression");
 96   return const;
 97   )
 98
 99 const(val) int val; {
100   immed();
101   outdec(val);
102   nl();
103   )
104
105 const2(val) int val; {
106   immed2();
107   outdec(val);
108   nl();
109   )
110
111 constant(lval)  int lval[]; {
112   lval=lval+3;
113   *lval=1;        /* assume it will be a constant */
114   if (number(++lval)) immed();
115   else if (pstr(lval)) immed();
116   else if (qstr(lval)) {
117     *(lval-1)=0; /* nope, it's a string address */
118     immed();
119     printlabel(litlab);
120     outbyte('+');
121     )
122   else return 0;
123   outdec(*lval);
124   nl();
125   return 1;
126   )
127
128 number(val)  int val[]; {
129   int k, minus;
130   k=minus=0;
131   while(1) {
132     if(match("+")) ;
133     else if(match("-")) minus=1;
134     else break;
135     )
136   if(numeric(ch)==0)return 0;
137   while (numeric(ch)) k=k*10+(inbyte()-'0');
138   if (minus) k=(-k);
139   val[0]=k;
140   return 1;
141   )
142
143 address(ptr) char *ptr; {
144   immed();
145   outstr(ptr+NAME);
146   nl();
147   )
148
149 pstr(val)  int val[]; {
150   int k;
151   k=0;
152   if (match("'")==0) return 0;
153   while(ch!=39)    k=(k&255)*256 + (litchar()&255);
154   ++lptr;
155   val[0]=k;
156   return 1;
157   )
158
159 qstr(val)  int val[]; {
```

(Continued next month)

Interrupts and CP/M

by Alan Bomberger

Alan Bomberger, 3721 Starr King Circle, Palo Alto, CA 94306.

How many times have you waited impatiently to enter that last command before heading to the dinner table? Computer systems without buffered input can be frustrating to use. This article illustrates the use of interrupts to provide fully buffered console input to CP/M (i.e., "type ahead").

Interrupts provide a mechanism for external signals to alter the function of a computer. Control is accomplished because an interrupt causes the computer to execute instructions in an order different from the order specified by the programmer. Although many microcomputer mainframes have provisions for interrupts, most operating systems (in particular CP/M) do not make use of interrupts and may, in fact, be designed to make the use of interrupts difficult. I have a NorthStar Horizon and wanted to use interrupts to control the console input. With proper modifications to the BIOS this would allow me to type data during program execution with the data being buffered in the memory of the computer until needed by either CP/M or my applications.

Z80 Interrupts

The Z80 and 8080 CPUs share a common interrupt mode; it is this mode I will discuss here. In this mode, an interrupt signal causes one of eight RST instructions to be processed by the CPU as its next instruction, regardless of which instruction is next in the normal flow of the program. The correct RST instruction is determined by hardware logic external to the CPU and depends on the particular interrupt line on the S100 bus that is active. The RST instruction causes the CPU to CALL a fixed location in the memory of the computer. RST 0 causes a CALL to location 0, RST 1 causes a CALL to location 8 and so forth. All interrupts can be disabled under program control through the use of a DI instruction. This instruction causes all interrupts to be ignored by the CPU until an EI instruction is executed and is used by programs when they cannot be interrupted (such as timing loops).

There are several important considerations for programs written to run with interrupts enabled and for interrupt subroutines. Programs written for an interrupt environment must have extra stack space available for use by the interrupt subroutine and must not have any timing or sequence-dependent code that is not protected by a DI-EI sequence. This latter condition extends beyond mere timing loops because the programmer must remember that the interrupt routine may completely alter the status of the computer and even cause a restart of the operating system. A particular example of this problem will be illustrated later.

Interrupt subroutines must be written with great care. Unless it is the specific purpose of an interrupt routine to cause a radical change in the flow of a program (such as causing a restart), no register or status information may be changed by the interrupt subroutine unless it is restored before the routine exits. Because interrupts are not synchronized with the executing program, it must be assumed that the interrupt will occur at the worst possible moment; when all registers and status bits are important to the executing program.

The Problem

I wanted to have an interrupt routine to input all data from the console of the computer and to buffer this input in memory until the console read functions were called by the application or the operating system. This part of the problem proved to be easy. A small buffer (128 bytes) is allocated in the BIOS region of the CP/M system and an interrupt subroutine uses this buffer in a circular fashion (when all 128 bytes are used, the first byte is reused). The Jump vector in the BIOS is altered so that the STATUS and INPUT addresses pass control to new subroutines in the interrupt handler.

The NorthStar Horizon can be modified via "jumpers" to cause an interrupt when console keys are struck. However, this interrupt, unlike the timer interrupt, cannot be independently disabled via software. Any time interrupts are enabled and the correct "jumper" is in place, the console will cause an interrupt. The timer interrupt has an independent control, via logic on the motherboard, that makes it possible for applications to turn it on and off without affecting other interrupts. Even though CP/M and the BIOS system allow interrupts in general, the timer interrupt is not specifically enabled unless an application program enables it. Since this control is not available for console interrupts, typing at the console would always cause an interrupt even if no application was ready to handle the interrupt. The NorthStar DOS system runs with interrupts enabled and, if the console interrupt "jumper" were added, I would have to modify the NorthStar DOS system to understand console interrupts.

It was for this reason that I chose to use the timer interrupt to simulate console interrupts. Each time the timer interrupt occurs the console is "polled" for input, and if a key has been struck the character is read into the buffer. The timer interrupt can still be used to time events and the console can still be read in the normal fashion without interrupts, so no changes need be made in the NorthStar DOS system.

The Design

Three new functions are provided by the interrupt-driven console routines. The first is buffered input ("type ahead"), the second is a fast abort (reboot of CP/M) and the third provides for flushing the input buffer. Each time a character is read it is checked to see if it is a Control-C. When a Control-C is detected the input buffer is cleared and a JMP 0 instruction is executed to cause a warm boot of CP/M. With this feature a program that gets stuck in a loop can be terminated without reaching around the back of the NorthStar for the RESET switch. When the input character is an ESC, all input but the ESC is removed from the buffer, and the ESC alone is left for the application or operating system to read.

Two pointers are kept by the console routines, and these two pointers control the flow of data into and out of the circular buffer. An input pointer is incremented every time a character is put into the buffer. When this pointer reaches the end of the buffer it wraps around to the beginning. An output pointer is incremented every time a character is read from the buffer, and it, too, wraps to the beginning when it reaches the end. The STATUS subroutine compares the two pointers and returns a status of "no character present" when the pointers are the same, and a status of "character present" when the pointers are different. Both pointers wrap around, and it is possible to get a

false status if exactly 128 characters are typed in advance of any read and to lose data if more than 128 characters are typed in advance.

An initialization routine that sets up the interrupt address at location 0020 and enables the timer interrupts is called during the cold boot.

Problems

Many computers, in particular the NorthStar Horizon, do not have DMA (direct memory access) disk controllers. This means that disk I/O takes place via some CPU instructions that must occur at a fixed rate. Any variation in the rate these instructions are executed will cause erroneous data transfers to and from the disk. For this reason, interrupts are disabled during data transfers. The NorthStar disk controller requires that the BIOS wait until the correct sector is under the read heads, and interrupts must be disabled during this time so that the data transfer can begin immediately after the correct sector is detected. If interrupts were enabled during this period it is possible that data at the beginning of the sector might be missed.

Characters typed during periods when interrupts are disabled are lost and it is desirable to minimize the length of these "dead" periods. If interrupts are disabled for no longer than a single disk revolution, as described above, there is little trouble, but if interrupts are also disabled during disk seeks, many typed characters can be lost. This was the case in the version of BIOS that I received as part of my CP/M package. This problem is solved only with a complete rewrite of the BIOS. An important aspect of interrupt programming must be considered in this rewrite, and I will take some time to explain it.

With many disk controllers it is the responsibility of the software to keep track of the seek position of the disk drive heads. A position counter is incremented each time the heads are stepped towards the hub and decremented each time the heads are stepped away from the hub. A comparison of the desired track position and the counter value determines the direction the heads must be stepped and the number of steps that must be made. It is important that the heads are always at the position indicated by the counter, for if they aren't, data will be read from the wrong track. If interrupts are allowed during all of the seek process, the head position and the counter become asynchronized when a Control-C is detected by the input interrupt routine between the time that the counter is updated and the time that the heads are stepped to their new position. Interrupts must be disabled during these two operations to ensure that they both happen together. If interrupts are enabled after each step operation, characters may be typed while the heads are moving.

Many applications (including the CCP of CP/M) test the status of the console during execution and alter their function if a character is typed. CCP aborts any command it is executing if a character is typed. This would entirely defeat the purpose of "type ahead," so CCP must be altered. Since I am using a special version of CCP written for the Z80 and have the source, I was able to modify CCP and eliminate this problem. Other applications posed other problems. The text editor I use assumes that any characters typed while it is computing (rewriting the screen or searching for strings) should be discarded; the effect is to have every other character used as input with alternate characters being discarded. Several hours were spent locating the offending code and patching it out with DDT. The CP/M assembler discards any typed character when it exits, and the effect is that one character is lost if any typing is done while ASM executes. I have not attempted to locate this problem.

Other programs routinely discard any input typed while they are executing. This annoying problem is solved only by habitually beginning any typed-ahead commands with Control-X. The Control-X, if read by the BDOS, causes the current line to be deleted and is harmless.

The worst problem is posed by an application that uses several of the RST instructions to execute subroutines. When this application is loaded, all of the locations from 0010 to 003F are overwritten and the next timer interrupt causes random branches in the application program. I have implemented special entry points in the BIOS to change the mode of operation of the console to interrupt-drive or standard.

Conclusions

I undertook a grand experiment to see if CP/M is flexible enough to allow buffered input. The results of this experiment are useful in that they illustrate many of the problems that must be considered when programming in an interrupt environment. I am using my buffered input BIOS as the standard on my NorthStar Horizon, and I am completely satisfied with its behavior. The listing included here is for the console interrupt routine, console input and status portions of the BIOS.

■■J

(Listing begins on page 57)

(Continued from page 31)

POLL compiles statements which poll the console for operator interruptions while the compiler is running.

CMD_LINE compiles statements which obtain the run-time options from the command line. The Standard I/O Library function getarg (Listing #2) obtains requested command-line arguments. It may require modification to work with your system. If CMD_LINE is not defined, then the compiler will prompt for each option.

PDS includes statements which identify files containing relocatable code for the second, third, and fourth parts of the compiler. These are unique to the PDS assembler/loader.

LINK implies that compiler output will be used with a relocatable assembler and linking loader. It compiles statements for declaring "extern" globals as external references and all other globals as entry points. In this case, multi-part programs cannot be combined at assembly time and the beginning label option ("-b#" switch) is not available.

COL causes labels in the output to be terminated by a colon.

TAB causes a character having the numeric value defined for TAB to be placed immediately before op codes in the output. Without TAB, nothing is inserted before op codes.

UPPER compiles statements which cause symbols placed into the symbol table to be converted to upper case. If your assembler does not require this, then disable the definition of UPPER.

PHASE2 refers to compile runs using this compiler. It causes lines which could not be handled by the original compiler to be included and alternate, old-style, lines to be excluded.

SEPARATE implies that the compiler is to be compiled in parts rather than all together. In this case, separate compile runs should be directed to each of the files cc1.c, cc2.c, cc3.c, and cc4.c. These, in turn, include subordinate files; e.g., cc11.c, cc12.c, and cc13.c (for part 1). If this symbol is missing, then cc1.c includes all of the subordinate files, and the files cc2.c, cc3.c, and cc4.c are not used.

NOCCARGC is a run-time option telling the compiler not to generate code for passing argument counts to called functions. This results in smaller, faster programs when it is known that there will be no calls to the run-time routine CCARGC.

HASH causes the compiler to employ a hash-search algorithm for macros and global symbols. Otherwise a serial search, requiring less memory, will be used.

Four symbols permit the user to determine which, if any, of the new language statements are to be supported by the compiler. Leaving them out may be required in order to make the compiler small enough to compile itself in one gulp on a 48K machine.

STDO controls the "do" statement. STFOR controls the "for" statement. STSWITCH controls the "switch," "case," and "default" statements. STGOTO controls the "goto" statement. These defines also associate a numeric value with each statement; the compiler uses this to determine if the last statement in a function is a "return."

OPTIMIZE causes the peephole optimizer to be included.

Availability

This version of Small-C is copyrighted as a revision to Ron Cain's original work. It is, nevertheless, available to the general public for use without formal restrictions. Take it, use it, copy it, modify it, and give it away as you please. If you capitalize on it for profit, I ask only that you compensate the principal authors in a manner that seems fair to you. Machine-readable copies are available on North Star and 8-inch single-density CP/M diskettes. CP/M versions contain just the source code as listed here and are available for $25. DOS versions with Small-VM, Small-Shell, and documentation cost $45. My software tools (written in Small-C for use with Small-VM) are also available, with documentation, for $45.

I will be happy to work with *DDJ* to establish a distribution scheme designed to make this compiler generally available at a nominal cost. If you can assist by adapting it to other operating systems, then let me know or write *DDJ*.

Editor's Note: Due to the popularity of DDJ *back issues, especially ones pertaining to Small-C, those which are referenced in this article have been sold out, and are not available for purchase. They will be made available in annual* DDJ *bound volumes.*

DDJ

(Listing begins on page 28)

Interrupts and CP/M

(Text begins on page 54)

```
;
;       console interrupt routines
;
int:
        di                      ; disable interrupts cause fool with pointers
        push    psw
        in      3               ; see if character present
        ani     02h             ; check status
        jz      intr1           ; nothing here reset clock and exit
        push    h               ; save some more stuff
        lxi     h,conbuf        ; base of buffer
        lda     inb             ; offset
        push    d               ; save for address calculation
        mvi     d,0
        mov     e,a             ; the offset
        dad     d               ; the new pointer
        pop     d
        inr     a               ; bump pointer
        cpi     080h            ; see if wrap
        jc      int1            ; still under limit
        mvi     a,00h           ; reset to beginning
int1:
        sta     inb             ; set input pointer
        in      2               ; get character
        ani     07fh            ; mask for CP/M
;
;       check for special characters
;
        cpi     03h             ; control c
        jnz     noboot          ; boot?
        xra     a               ; smash buffer
        sta     inb
        sta     outb
        jmp     0               ; boot it
noboot:
        cpi     01bh            ; escape
        jnz     stashit         ; put into buffer
        push    psw
        lxi     h,conbuf        ; to beginning
        xra     a
        sta     outb            ; set to beginning
        inr     a               ; reset input so one character in buffer
        sta     inb             ; zap all but one
        pop     psw
stashit:
        mov     m,a             ; put character in buffer
        pop     h
intr1:
        mvi     a,050h
        out     6               ; reset clock flag
        pop     psw
        ei
        ret
status:
        lda     outb            ; compare input and output pointers
        lxi     h,inb
        cmp     m               ; are they different
        mvi     a,0ffh          ; in case they are
        rnz                     ; there is a character
        xra     a               ; there isn't
        ret
input:
        call    status          ; check status
        jz      input           ; loop
        push    d
        mvi     d,0             ; offset
        di                      ; protect pointers from interrupts
        lda     outb            ; output buffer pointer
        lxi     h,conbuf        ; base of buffer
        mov     e,a             ; build
        dad     d               ; build new address
        pop     d
        inr     a               ; bump pointer
        cpi     080h            ; see if wrap
        jc      input1          ; no
        mvi     a,00h           ; yes reset to beginning
input1:
        sta     outb
        mov     a,m             ; fetch character
        ei                      ; allow interrupts
        ret
inb:    db      0               ; input displacement
outb:   db      0               ; output displacement
;
;       initialization routine must be called to start
;
clinit:
        mvi     a,jmp
        sta     020h            ; set up clock interrupt
        lxi     h,int           ; place to go
        shld    021h
        mvi     a,0c0h
        out     6               ; arm clock
        mvi     a,050h          ; reset flag
        out     6
        ret
intoff:
        mvi     a,040h
        out     6
        lxi     d,const
        lxi     h,confix
        lxi     b,6
        db      0edh,0b0h
        ret
inton:
        mvi     a,0c0h
        out     6
        lxi     d,const
        lxi     h,conint
        lxi     b,6
        db      0edh,0b0h
        ret
conint:
        jmp     status
        jmp     input
confix:
        jmp     hs0sis
        jmp     hs0sin
;
;       set up transfer vector
;
        org     bios+0200h+6
        jmp     status
        org     bios+0200h+9
        jmp     input
        org     usetion
        call    clinit
;
        org     0ff80h
conbuf: ds      128
```

End Listing

A Simple Vector Generation Algorithm

Editor's note: Regular readers of DDJ *know that we rarely publish articles of this nature without accompanying source code. It is our opinion that an exemplary listing can often do as much, or more, to convey a point as the best of documentation. Unfortunately, we have been unable to reach Mr. Michalski at his Poland address. He signed his original cover letter "Sincerely, permanent reader of* DDJ*." We sincerely hope he is just that. – MDO*

Microprocessors are cheap devices, very useful for controlling processes. Drawing pictures and characters is the main task of the graphics display. The microcomputer can be used to generate vectors from given coordinates of the endpoints, *viz.* generate all points between two given endpoints in such a way that the the vector seems to have good continuity and linearity.

In the first part of this paper we shall try to find a simple and fast algorithm for vector generation – useful for microprocessor computations. In the second part we shall give an example of its implementation for 8-bit microprocessors.

Description of the Algorithm

Let (x_s,y_s) and (x_f,y_f) denote two given endpoints of a vector. Any arbitrary endpoint $(x,y) \in R_2$ belongs to this vector if it satisfies the following formula:

$$/i/ \qquad y-y_s = \frac{\Delta y}{\Delta x}(x-x_s) \quad \text{and} \quad x_s \leqslant x \leqslant x_f \quad \Delta x \neq 0$$

$$\text{or} \qquad x-x_s = \frac{\Delta x}{\Delta y}(y-y_s) \quad \text{and} \quad y_s \leqslant y \leqslant y_f \quad \Delta y \neq 0$$

where $\Delta x := x_f - x_s$, $\Delta y := y_f - y_s$

First, let us consider the vector generation when:

$$/1/ \quad \Delta x > 0 \quad \text{and} \quad \Delta y \geqslant 0,$$
$$/2/ \quad \Delta x \geqslant \Delta y.$$

Our considerations would be similar in the other seven cases corresponding to different signs of Δx and Δy as well as relations between their absolute values.

Let us try to transform the formula /i/ to an equivalent one. According to /1/ we have:

$$(y-y_s)\Delta x = (x-x_s)\Delta y \quad \text{and} \quad x_s \leqslant x \leqslant x_f$$

Coordinates of all points are integer numbers so: $y-y_s = k$ and $x-x_s = m$ where $k = 0,1,\ldots,\Delta y$ and $m = 0,1,\ldots,\Delta x$. This gives:

$$/ii/ \quad k\Delta x - m\Delta y = 0$$

For good linearity and continuity of the vector we construct the sequence of the points $\{(x_i,y_i)\}_{i=0,1,\ldots,\Delta x}$ where:

$$x_o = x_s, \ldots, x_i = x_{i-1}+1, \ldots, x_f$$
$$y_o = y_s, \ldots, y_i = y_{i-1}+e_i, \ldots \qquad e_i = 0,1$$

and $\{e_i\}$ is defined in the following way:

$$e_i = \begin{cases} 0 & \text{if } |F_i(0)| < |F_i(1)| \\ 1 & \text{otherwise} \end{cases}$$

where $F_o(e) = 0$, $F_i(e) = F_{i-1}(e_{i-1}) - e\Delta x + \Delta y$ $e=0,1$ and $i=0,1,\ldots,\Delta x$. This means that for every m, $m=0,1,\ldots,\Delta x$ we want to find such an integer k that the absolute value of the expression $k\Delta x - m\Delta y$ is minimal. Figure 1 shows the flowchart of the algorithm for the above case. Similar algorithms can be written for the other cases.

The algorithm is hardware independent. Machine-specific code will deal simply with incrementing/decementing by 1, adding and comparing two numbers, and branching and marking the points onto the display. The algorithm can be used for every microcomputer system with a CRT as an input/output device.

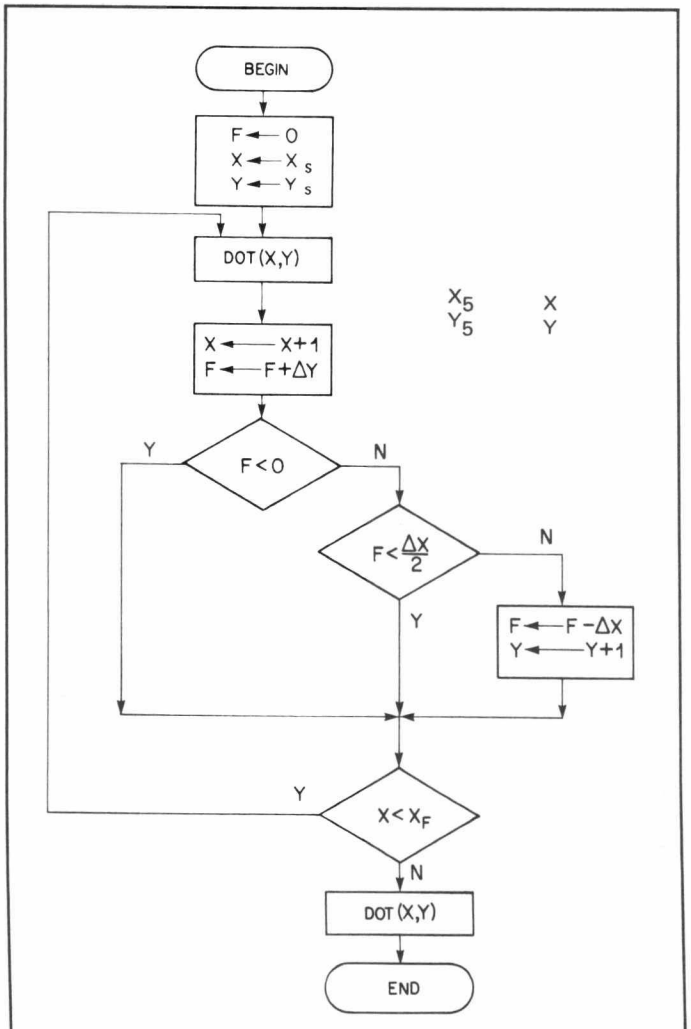


Figure 1.
The flowchart of the algorithm.

by Marek W. Michalski

Marek W. Michalski, Technical University of Warsaw, Institute of Mathematics, 1 Jedn. Robot. Sq., Warsaw, Poland.

Implementation

The algorithm was implemented for microcomputers based on the 8-bit microprocessor MC6800. Due to the nature of the displays, the values of the coordinates of the display space were integers and limited to the range: $0 \leqslant x \leqslant 22FH$ and $0 \leqslant y \leqslant 1AFH$. Data were sent from the host computer to our microcomputer every 20 ms. The algorithm generated vectors between two consecutive transfers of data for almost all cases – so only a small buffer was needed.

Generally, the vector generation program was written in the following way:

Step 1. Choosing the case by examination of the signs of Δx and Δy and the relation between their absolute values,

Step 2. Generation of the vector according to the described algorithm.

Using the HP1611A Logic State Analyzer the time intervals of the longest generated vectors were measured. The time intervals were as follows:

Vector no.	1	2	3	4	5	6
Time (ms)	36.3	36.4	23.8	22.6	23.8	22.7

Acknowledgement

I would like to thank Dr. G. Hamano and Mr. M. Noguchi for their fruitful advice on technical matters during my work at Matsushita's Wireless Laboratory in Osaka.

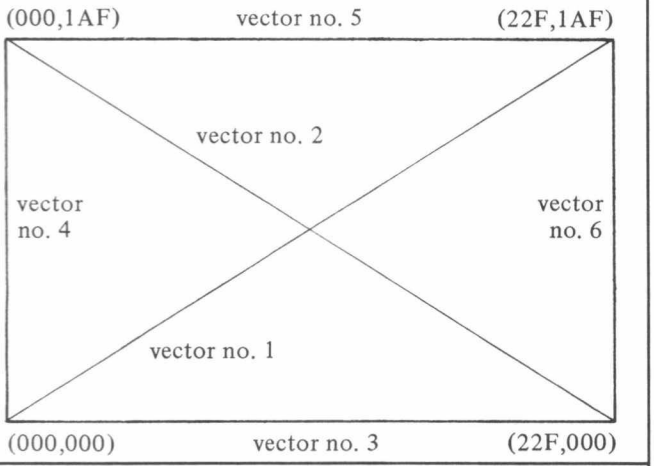


Figure 2.

Vector graphics test pattern. The vector test generator produces a six-segment rectangle with the diagonals at the extremes of the display space. In the figure, hexadecimal coordinates of the endpoints are shown in parentheses.

Guest Essay

5th Generation Computers

If one were to walk into a Computerland store in 1992, what could one expect to be able to purchase?

The fifth generation computer of the early 1990s will not be a mainframe in the conventional sense; rather, it will be a "supermicro." It will be based upon a 64-bit main microprocessor running about 110 megahertz with a 32-bit microprocessor for I/O. Both will still be made of silicon, incidentally, though by 1992 gallium arsenide circuits should be spreading throughout the industry.

Notice how I have not mentioned using an array of 16- or 32-bit microprocessors functioning simultaneously, simply because I do not see the problem of distributed array processing as having been solved by 1990. Although Illiac IV was a spectacular early achievement in using a large array of processors in parallel, the von Neumann architecture will nevertheless be with us for some time to come. Distributed array processing should definitely appear by the year 2000, though, in which case a language similar to Ada (which is itself a descendant of Concurrent Pascal) will be in vogue, since such languages are in theory capable of controlling a number of different processors "simultaneously."

The fifth generation computer will have 4.096 megabyte EEPROM memory chips, with a total of a little over 32 megabytes of internal storage. Bubble memories could be used instead, though if they do become popular they will probably be of Japanese manufacture. IBM has been working on a computer (to be ready in 1985 or thereabouts) utilizing a Josephson Junction processor and 15 megabytes of bubble memory. The machine will be of incredibly small size: a cube about 15 centimeters on a side, but the entire device must be cooled to superconducting temperatures, which will require support equipment about the size of a desk.

External storage will be in the form of optical laser disks capable of recording through some kind of diffraction process. The disks will be small (not larger than 10 inches in diameter), easily transportable, and will have a capacity of between 10,000 and 15,000 megabytes. Since the more advanced artifical intelligence programs of the early 1990's will exhaust even 32 megabytes of memory, a fast virtual memory system will be developed utilizing these optical storage disks.

The display will be flat, wide-screen, and in color, and probably based upon Langmuir-Blodgett film technology. They shall be wide enough to display a menu *and* two or three sections of a program across the width of the display while editing in data processing, *or* three or four standard 8-1/2 x 11 inch pages simultaneously while doing word processing. The editor used on the programs will be powerful enough to do both data processing and word processing, and will probably be some kind of EMACS/WYLBUR/Yale Editor hybrid with a little Xerox "mouse" replacing the cursor direction keys.

Voice entry will be available as an option, but it will still take up an inordinate amount of processing time, power and memory. Voice entry will be popular among unskilled word processor personnel and older executives. However, programmers may never get the knack of dictating large systems software (to say the least) and it will be difficult to pry them away from the control offered by a standard keyboard with a fast, command-driven editor.

Specially designed, flexible, motorized chairs will be offered to place the operator at the correct height and distance from the keyboard. Some of the more elaborate models will be offered as a sort of peripheral with larger computers so that the computer, keyboard, display and chair will be built into a single framework, giving the device the appearance of a wide-screen flight simulator.

The biggest revolution will come in terms of software. "Actor-based" languages such as Smalltalk will allow programs to alter themselves to the user's wishes. If various present-day languages were embedded in such an actor language, then strange, new, customized languages for specific purposes could be generated on demand, merely by the user having a "conversation" with the larger host AI/actor program. Pascal's inferior string and file handling capabilities, for example, could be improved by instructing the host actor language to start with a Pascal "foundation," then importing a few functions (actors) from SNOBOL 4 and COBOL.

This language of the future will be close to its own operating system (like C and Forth), will have no "reserved" words, will be extensible, and the language's own procedures will be as accessible to its programs as its data structures, á la LISP.

The language will have extremely high-level subroutines resembling natural English, and a program's self-alterations as a result of communicating with a large number of users (its "learning" process, if you will) will allow natural language processing to finally come of age. We will be on the verge of attaining what I. J. Good in the early 1960s called the "ultra-intelligent machine."

This actor-based language will probably be embedded not in one of the LISPs *per se*, since LISP will have to be given some kind of structured treatment for the optimum usage of the optical virtual memory disk system. I think the language closest to what I have in mind is something called LISP/ALGOL, which was developed at the University of Grenoble back in the 1960s.

By the early 1990s, the advanced, compact hardware and artificial intelligence software described will start to be incorporated within robots. Then we'll probably let the machines themselves design *really* new *sixth* generation equipment!

by Richard Grigonis

Richard Grigonis, Children's Television Workshop, One Lincoln Plaza, New York, NY 10023.

DR. DOBB'S CLINIC

by Jan-Henrik Johansson, Guest Intern

Argc and *Argv* for Small-C

After having worked for some time with UNIX at my office and with The Code Works[tm] version of Small-C at home, I started to feel that, although useful, Small-C nevertheless lacked some features I regularly used on UNIX. One was the capability to define standard input and output at the moment I run my program rather than at the moment(s) I wrote it. Another equally important one is the capability to pass arguments to the program from the command line.

Although most implementations of C actually support these two features, the Small-C package distributed by The Code Works for CP/M does not include them (they have more expensive versions that do, however). I guess there are many of you out there who use Small-C and may therefore be interested in the programs in Listing 1 (page 64) which gives your Small-C both of these features. The programs are written in Small-C and fit the CP/M package distributed by The Code Works without any changes, but they should run with very few changes on other operating systems. You only have to know where your operating system puts your command-line buffer.

Command-line Syntax

When you invoke your program, standard input and/or output is redirected by redirection instructions given anywhere, and in any order, on the command line. To be more precise: to redirect stdin, enter "<" followed by the filename of your CP/M file, or by the device number of the peripheral device you want to assign. Device number 0 is the console; device number 1 is the printer. If the file or the device does not exist you will receive an error message and the program will terminate. To redirect output, enter ">" followed by the filename or the device number. If the file does not exist, it will be created; otherwise, it will be written over.

Any character strings in the command line not preceded by "<" or by ">" are considered arguments. They are stored by the program for later use. One or several spaces are delimiters. The program accesses these arguments using the getarg() function. In short, the command-line syntax is:

progname [<*stdin*] [>*stdout*]
[*argument1*] . . .

Accessing the CP/M Command Line

The routine setargs() (lines 34-113 in Listing 1) contains the code which accesses the CP/M command-line buffer and interprets it (all in Small-C!). The format of the CP/M command-line buffer is: a one-byte counter followed by as many bytes holding the command line, starting with the first space after the program name and ending with the last character preceding the carriage return. The CP/M buffer is usually located at 80H (128 decimal). The variable "count" points to it (line 44 in Listing 1). Notice that CP/M does not retain the name of the program you invoked. In my view, this is rather unfriendly; in any case, it does not match the UNIX C convention that argument 0 should contain the name of the invoked program. In order to make the numbering of the command-line arguments compatible with UNIX and make my programs more portable, I decided to provide an asterisk as argument 0 (line 47 in Listing 1).

Accessing the Arguments

In *The C Programming Language* by Kernighan and Ritchie, the processing of command-line arguments is described in the following way (page 110): "When MAIN is called to begin execution, it is called with two arguments. The first (conventionally called argc) is the number of command-line arguments the program was invoked with; the second (argv) is a pointer to an array of character strings that contain the arguments, one per string." To make argc and argv the arguments of main() would require a change to the Small-C compiler; fortunately there is a simpler way. It is sufficient to make them, together with stdin and stdout, external. Pointers to pointers are not allowed in Small-C, so argv is simply an integer array, the values of which are pointers to the individual arguments. To finally get to the arguments in a Small-C program, you call getarg(), whose interface is described in Figure 1.

An Example

Figure 2 is a complete example showing how to use setarg() and getarg(). Instead of calling setarg() from main() as in the example, the call could be made from CCGO in c80lib so you would not have to remember to initialize. Also, getarg.c could of course be added directly

to c80lib. Notice, by the way, that CP/M gives your arguments back in upper-case only. Notice also that your output routines (fputc in Figure 2) must know about stdin and stdout – this is where the redirection finally takes place! ▶▶J

(Listing begins on page 64)

getarg() function:

n = getarg(i,s,l,argc,argv)

n	EOF or a pointer to the found argument string
i	is the number of the desired argument
s	string into which getarg will return the argument
l	maximum length of string to be returned
argc	total number of arguments
argv	array of pointers to arguments

Figure 1.
The interface to the getarg function.

```
A>
A>type testgeta.c
#include getarg.c
main() {
int i;
char *c, s[20];

setargs();

i=0;
while(getarg(i++, s, 20, argc, argv) != EOF) {
        fputc(CR, stdout);
        c=s;
        while(*c) fputc(*c++, stdout);
        }
}
fputc(c, fd) char c; int *fd; {
        if(fd == CONSOLE) putchar(c);
        else if(fd == PRINTER) syslout(c);
        else putc(c, fd);
        return NULL;
        }
syslout(c) char c; {
        cpm(5, c);
        if(c==CR) cpm(5, 10);
        }

#include b:c80lib


A>testgeta >1 send this to the printer

*
SEND
THIS
TO
THE
PRINTER
```

Figure 2.
A demonstration of the use of setargs, getargs, and I/O redirection.

ARGC and ARGV (Text begins on page 62)

Written for The Code Works' Small-C compiler

Listing One

```
 1 #define EOF -1
 2 #define NULL 0
 3 #define YES 1
 4 #define NO 0
 5 #define CR 13
 6 #define SPACE 32
 7 #define BELL 7
 8
 9 #define CONSOLE 0
10 #define PRINTER 1
11 #define SERIALA 2
12 #define SERIALB 3
13
14 int stdin, stdout, stderr, stdlist;
15 int argc, argv[10];
16
17 getarg(n,s,size,argc,argv)
18    int n; char *s; int size, argc, argv[]; {
19 char *str;
20 int i;
21
22    if(n < 0 | n >= argc) return EOF;
23    i=0;
24    str=argv[n];
25    while(i<size) {
26            if(str[i]==NULL) break;
27            s[i]=str[i];
28            i++;
29            }
30    s[i]=NULL;
31    return s;
32    }
33
34 setargs() {
35
36    stdin=stdout=stderr=CONSOLE;
37    stdlist=PRINTER;
38
39    int error;
40    int i, j;
41    char args[80], fname[25];
42    char *count, *ptr, *lastc;
43
44    count=128;  /* CPM command buffer */
45    ptr=count+1;
46    lastc=ptr+*count;
47    args[0]='*'; args[1]=NULL;
48    j=2;
49    argv[0]=args;
50    argc=1;
51
52    error=NO;
53
54
55    while (++ptr < lastc) {
56            if(*ptr == SPACE)
57                    continue;
58            else if(*ptr == '<' ) {   /* redirect input */
59                    while(*++ptr == SPACE) ;
60                    if(ptr >= lastc) break;
61                    if(*ptr == '0') {
62                            stdin=CONSOLE; ++ptr; }
63                    else if(*ptr == '1') {
64                            putchar(CR);
65                            puts("You cannot assign printer as input");
66                            error=YES;
67                            }
68                    else {
69                            i=0;
70                            while(*ptr != SPACE & ptr < lastc)
71                                    fname[i++]=*ptr++;
72                            fname[i]=NULL;
73                            stdin=fopen(fname,"r");
74                            if(stdin == 0) {
75                                    putchar(CR);
76                                    puts(" Can't open ");
77                                    puts(fname);
78                                    error=YES;
79                                    }
80                            }
81                    }
82            else if(*ptr == '>') {    /* redirect output */
83                    while(*++ptr == SPACE) ;
84                    if(ptr >= lastc) break;
85                    if(*ptr == '0') {
86                            stdout=CONSOLE; ++ptr; }
87                    else if(*ptr == '1') {
88                            stdout=PRINTER; ++ptr; }
89                    else {
90                            i=0;
91                            while(*ptr != SPACE & ptr < lastc)
92                                    fname[i++]=*ptr++;
93                            fname[i]=NULL;
94                            stdout=fopen(fname,"w");
95                            if(stdout==0) {
96                                    putchar(CR);
97                                    puts(" Can't open ");
98                                    puts(fname);
99                                    error=YES;
100                                   }
101                           }
102                   }
103           else {   /* argument */
104                   argv[argc]=args+j;
105                   argc++;
106                   while(*ptr != SPACE & ptr < lastc)
107                           args[j++] = *ptr++;
108                   args[j++]=NULL;
109                   }
110           }
111
112   if(error == YES) exit(BELL);
113   }
114

A>
```

End Listing

by Michael Wiesenberg

Of PAC RAT and Other Hardware

H-1000 from Technical Micro Systems Inc. is a Z-80/8086 upgrade for the H89/Z89. It replaces the CPU, requiring no modifications, with a software-selectable 2 and 4 MHz clock and an 8 MHz 8086. In Z-80 mode, the H-1000 runs all present Heath/Zenith HDOS, CP/M-80, or Pascal software; in 8086 mode, it is compatible with Zenith's Z-100 and the IBM PC with MSDOS, CP/M-86, or both. The board comes with 128K RAM, expandable to 1 megabyte. A RAM disk emulator copies an entire disk into RAM for instant access. For $1495 you get the board, five input/output slots, diagnostic software, and MSDOS or CP/M-86. Multi-user and multi-task operating systems are also available. **Reader Service No. 469.**

Six megabytes of on-line random access storage, with removable cartridges, for $480? **PAC RAT** from DAMCO is a *r*andom *a*ccess *t*ape drive that uses two inexpensive (they say, though they quote no price) removable magnetic tape cartridges. The package is the same size and shape as a minifloppy drive, with standard minifloppy power requirements and controller interface so it can plug right into existing systems. You'll have to provide a box for the unit, and mount it yourself, and you may have to add a fan, depending on its orientation. Transfer time is 250,000 bits per second, latency is typically 400 milliseconds, and random access 740 milliseconds. Reliability specifications, they state, are "unknown at this time." **Reader Service No. 431.**

Hand-held computers can be given the capabilities of much larger systems with the **DataShuttle Peripheral** from F/A Technology (formerly Friends Amis), developers of the HHC Hand-Held Computer System marketed by Panasonic and Quasar. The DataShuttle is a "sleeve" that fits around an HHC, and provides up to 40K non-volatile CMOS RAM, 96K ROM, a direct-connect modem with automatic dialing, an acoustic coupler, real-time clock, a cassette tape controller, RS-232 and parallel ports, Forth and BASIC compilers and interpreters, and word processing software. Peripherals include an EPROM burner, bar code reader, portable printers, and TV adapter. System prices start at $380. **Reader Service No. 441.**

Don't throw away that wornout printer ribbon! The Ink-A-New company will re-ink your cartridge for free (offer good until January 15) if you purchase a custom-designed **Ink-A-New Cartridge Re-Inker** for $22.50 plus $2.50 p&h. Since cartridges are not standard in design and dimensions, they need your old one to custom build a Rube Goldberg-like gadget through which you hand crank ribbons when their ink begins to fade, permitting virtually endless reuse of ribbons. They offer a three-week money-back guarantee on the $22.50, and you can still keep the re-inked cartridge. **Reader Service No. 449.**

Softer Wares

CP/M has not thus far been available for Radio Shack's hard disk, but Aton International has changed that by introducing **ATON CP/M 2.2** for TRS-80 Models II and 16. Partitioning can subdivide the hard disk into logical drives that simulate multiple drives, permitting backing up onto unused parts of the same disk. Data can also be backed up onto a different hard disk, or files as large as 8 megabytes can be backed up onto multiple floppies. Either TRSDOS or CP/M can be booted up, so that application programs formatted for either can be stored on the same hard disk, or ATON CP/M can bootstrap directly from the hard disk. The software costs $279; present ATON CP/M owners can upgrade to hard disk format for $125. **Reader Service No. 428.**

Digital Marketing offers **Active Trace** for MBASIC. Its SCOPE facility provides real-time information about running programs. Like MBASIC's TRON (trace on) facility, it displays line numbers, but also displays variables with their current values as encountered. Optionally, specific variables and program sections can be isolated for analysis. VREF cross references all variables, and GOREF cross references all GOTO and GOSUB statements. The documentation includes information on programming techniques to minimize errors and make them easier to find, and details on how to find the errors. Active Trace normally costs $125, but contact them for their special price to be offered during December only. **Grammatik**, the word processing facility that goes far beyond spelling dictionaries by actually checking your grammar and writing style and looking for overuse and misuse of certain key phrases, is also available for $150. **Reader Service No. 435.**

Third-party software for the IBM Personal Computer can be reasonably priced. **Info-Sort** from Info-Pros is a sort/select package they claim to be the fastest available. Reading and sorting one thousand 64-byte records on a 6-byte character key takes under 20 seconds. This assembly-language utility runs under PC-DOS (i.e., MSDOS) or interfaces with applications, and uses a control file to add or modify sort/select parameters. Keys (up to nine) can be strings, integers, and single- or double-precision numbers. The output can be a sorted copy of the input, a file or relative record numbers of the sorted input, or an index file that contains the sorted keys with the relative record number of each. Up to 64K of records, each with a length up to 1K, can be sorted, in ascending or descending order. Up to eight logical conditions can be used to include or omit records, and as many as 999 records can be skipped. Your IBM PC needs 64K, PC-DOS, one drive, an 80 x 25 monitor, and BASIC (for the demonstration and to modify control files). $145. **Reader Service No. 445.**

InterLync, from Starside Engineering, is a terminal emulator and file transfer package for the IBM PC. It communicates with mainframes, CP/M systems, APL timesharing systems, etc. Files sent to and received from CP/M computers and bulletin board systems use the "Xmodem" block transfer protocol, or you can configure in may ways an ETX/ACK line-oriented protocol. Baud rates are 110 to 9600, data words are seven or eight bits, with even, odd, or no parity. InterLync is controlled through various screens by the PC's programmable function keys. An APL option can be ordered to replace the graphics character PROM on either PC video board, and character-to-key mapping can be switched between ASCII and APL with one keystroke. You'll need 128K, one drive, and an asynchronous communications port. The standard version costs $100; add $75 for the APL option. **Reader Service No. 465.**

Three Forths

COLOR-FORTH, by Hoyt Stearns Electronics, for the TRS-80 Color Computer has a RAM version that requires 16K of RAM, and a ROM version that works with 4K of RAM, and can be installed in a disk controller or in a ROM cartridge. It comes with an editor and utilities. COLOR-FORTH has a trace feature, handles interrupts, maintains the

CPU carry flag for extended math, and supports bit manipulation, graphics, sound, task multiplexing, fast math, auto-repeat keys, control keys, disk and tape control, and links to BASIC routines. The RAM version is $58.95, while $123 gets you ROM plus RAM version. **Reader Service No. 471.**

FORTH-32, from Quest Research, is a complete software development system for the IBM PC. It uses 16- and 32-bit addressing modes; the interpreter "knows" at run time which mode was compiled. Sixteen bits are sufficient for programs that reside within 64K bytes; 32-bit addressing is used for larger programs. FORTH-32 is fig-FORTH compatible. For $100, the package includes base language, segment-sensing compiler, mixed-mode addressing interpreter, multi-mode verbs (reserved words), case statement, package builder utility, 8086 assembler, decompiler, screen editor, debugger, DOS interface, and graphics. A floating-point package that includes software for 8087 floating-point library and an 8087 assembler compatible with the 8086 assembler costs $50. **Reader Service No. 457.**

fig-FORTH-79, from Mark Manning, is fig-FORTH with double-number standard extensions for any 6502 system that uses Micro Technology Unlimited's CODOS. It comes with an editor, 6502 assembler, and utility screens, on 8-inch single-sided double-density disk for $145, or on cassette without the interface to CODS, for any 6502 system that reads either KIM or MOS Technology Standard recording formats, for $38. **Reader Service No. 453.**

One FORTRAN

Nevada FORTRAN for CP/M, from Ellis Computing, is both a subset and superset of ANSI 1966 FORTRAN. Extensions include IF . . . THEN . . . ELSE, TRACE debugging, COPY, seven-dimensional arrays, and random access to files. The compiler generates 8080 machine code on any CP/M system with a minimum of 32K. The diskette and 165 pages of documentation run $199.95. **Reader Service No. 439.**

TRACE, THUG, RAMS, and Other User Groups

TRACE, the Toronto Region Association of Computer Enthusiasts, a non-profit organization, publishes a monthly newsletter mailed free to members. Membership is $13 a year. TRACE just joined forces with THUG (the Toronto Heath Users Group). TRACE appears to be for users of all microcomputers. Write to TRACE, Box 6922, Station "A", Toronto, Ontario M5W 1X6.

RAMS, the Rochester Area Microcomputer Society, offers a monthly newsletter, *Memory Pages*, with membership, which runs from October to September, for $7.50, or $4 after April. RAMS regularly schedules experts from computer companies to speak and demonstrate new equipment. The entire group meets monthly, and CP/M and North Star "subgroups" meet between times. Rochester Area Microcomputer Society, Box 90808, Rochester, NY 14609.

Membership in **The Northwest Computer Society** is for personal and small business computer users in the Puget Sound area, and includes in its yearly $12 dues a subscription to *The Northwest Computer News*, a professional-looking paper also sold on newsstands. Northwest Computer Society, Box 31495, Seattle, WA 98103.

The **CSRA Computer Club** of Augusta, Georgia, holds meetings open to the public the third Thursday of each month. $12 yearly dues include a monthly newsletter. CSRA Computer Club, Box 284, Augusta, GA 30903.

The **Amateur Computer Group of New Jersey** is a non-profit educational corporation for users of all micros. The $14 yearly membership includes a subscription to the monthly *ACG-NJ News*. Amateur Computer Group of New Jersey, Box 319, South Bound Brook, NJ 08880.

DDJ

Letters *(Continued from page 9)*

It all got started when I was playing around with the graphing routine provided with JRT Pascal. When I attempted to use it my system would "hang." By "hang" I mean nothing seemed to be happening. I used the trace facility of JRT Pascal to isolate the problem to a file close within the graphing routine.

I tried file operations, including open and close, with other user programs and had no problem. So, I proceeded to try to isolate the problem further with DDT.

When I attempted to use DDT I discovered another problem. The sequence of commands was as follows:

DDT EXEC.COM	to load the JRT Pascal Executive
IPROGRAM.INT	to identify user program
G	to commence execution

When the executive attempted to open the user program file to load it into the system, the system would hang again.

Then I decided to try to determine whether this new problem was related to JRT Pascal or not. So, I tried DDT with a dump utility as follows:

```
DDT DUMP.COM
IDUMP.COM
G
```

The results were the same. When the dump program attempted to open and read itself the system hung.

From these experiences I hypothesized that I had two problems, one with JRT Pascal and one with DDT, or I had one problem with the operating system. I proceeded to attempt to trace instruction execution to isolate the problem still further.

I soon relearned that DDT does not trace BDOS or BIOS, and that is where the system hangs. I learned of a method for getting DDT to trace BDOS execution. I put a RST 7 instruction at a strategic location in the BDOS. When this instruction is executed, DDT breakpoints. Then tracing can be done. However, more problems arose to cloud the issue.

I had difficulty with RET instructions. It seems that DDT would not return to the proper instruction or the system would hang if I traced through a RET. I examined the stack and made another befuddling discovery. DDT messes it all up. I hypothesize that DDT puts trap addresses on the stack so it can regain control after a RET. I don't know for sure, though. Anyway, I discovered for sure that I could not trace through a RET.

I also discovered, by the way, that DDT will not allow overwrite of the stack with the "F" command. I tried this in a moment of desperation to see whether I had problems writing to memory at that location. No go. I could zap bytes on both sides of it, but not in it.

Next I chose to disassemble code and insert RST 7 instructions at the return locations from RET instructions and at jump instructions. Then I would give DDT the command to go. It would breakpoint, and I could examine the situation. If the system did not hang, I would proceed in the same manner through the next stretch of code.

Eventually, I isolated the hang-up to a small section of code in the BIOS. Since the BIOS is in ROM, I could not breakpoint in the BIOS, but I did put enough clues together to isolate the hang-up to a very small section which included the Z-80 EI, INI instructions and, at the end, a RETI instruction. Since the INI and RETI instructions are not 8080 instructions, I reasoned that they, especially the RETI, may cause a problem for DDT, but I don't know for sure.

Another thing I don't know is how this problem relates to the problem with JRT Pascal, if it does.

So, why did I write? And what do I want? Well, I don't hold out a lot of hope for finding the answers to my close problem with JRT Pascal or my problem with DDT, but maybe someone could at least enlighten me on the inner workings and methods of DDT. Who knows, maybe the real answer will pop up.

Of specific interest to me is what DDT does to the stack, what happens to RET instructions when tracing BDOS

```
A>JRTPAS2 MULTIPLY

JRT Pascal   ver 2.1

Copyright 1982 JRT Systems

0000  0001:     %LTRACE
0003  0002:
0006  0003:     PROGRAM MULTIPLY( OUTPUT );
0009  0004:     { 5 September 1982 }
000F  0005:     VAR TEMP,TEMP0 : REAL;
0015  0006:     BEGIN
0025  0007:       TEMP := 0.0;
002E  0008:       TEMP0 := 0;
004C  0009:       WRITELN( 'TEMP = ',TEMP );
00A5  0010:       WRITELN( '2.0*TEMP = ',2.0 * TEMP,'  2.0*0.0 = ',2.0 * 0.0 );
00FE  0011:       WRITELN( 'TEMP*2.0 = ',TEMP * 2.0,'  0.0*2.0 = ',0.0 * 2.0 );
0155  0012:       WRITELN( '2.0*TEMP0 = ',2.0*TEMP0,'  TEMP0*2.0 = ',TEMP0*2.0 );
017E  0013:       WRITELN( '2*0 = ',2*0,'  0*2 = ',0*2 );
01DB  0014:       WRITELN( '3.0*0.0 = ',3.0*0.0,'  0.0*3.0 = ',0.0*3.0 );
0238  0015:       WRITELN( '4.0*0.0 = ',4.0*0.0,'  0.0*4.0 = ',0.0*4.0 );
0295  0016:       WRITELN( '5.0*0.0 = ',5.0*0.0,'  0.0*5.0 = ',0.0*5.0 );
02F2  0017:       WRITELN( '3.5*0.0 = ',3.5*0.0,'  0.0*3.5 = ',0.0*3.5 );
034F  0018:       WRITELN( '4.3*0.0 = ',4.3*0.0,'  0.0*4.3 = ',0.0*4.3 );
03AC  0019:       WRITELN( '5.8*0.0 = ',5.8*0.0,'  0.0*5.8 = ',0.0*5.8 );
No errors detected
Module size = 948 dec bytes
End of compile for MULTIPLY

A>EXEC MULTIPLY
Exec  ver 2.1

TEMP =  0
2.0*TEMP =  2  2.0*0.0 =  2
TEMP*2.0 =  0  0.0*2.0 =  0
2.0*TEMP0 =  2  TEMP0*2.0 =  0
2*0 =  0  0*2 =  0
3.0*0.0 =  3  0.0*3.0 =  0
4.0*0.0 =  4  0.0*4.0 =  0
5.0*0.0 =  5  0.0*5.0 =  0
3.5*0.0 =  3.5  0.0*3.5 =  0
4.3*0.0 =  4.3  0.0*4.3 =  0
5.8*0.0 =  5.8  0.0*5.8 =  0

Program termination
```

(I suppose the answer is that it shouldn't be done), and what it does with Z-80 instructions in the go mode and in the trace mode (I suspect there's a difference).

If you or your readers can supply any of this or related information, I would appreciate it. Thanks for your interest.

Sincerely,
Monte Schmiege
707 Kilbourn Ave.
West Bend, WI 53095

DDJ

Index

This is an index to articles and programs, though some letters have also been included. References are of the form *bound volume number / page number.* The processor type or format, where relevant, follows the entry in brackets. Items are listed by author(s), processor type, and program type. Cross references are noted in parentheses following the entries. References separated by a comma are connected in some way; those separated by a semicolon are completely distinct. This is a cumulative index for *Dr. Dobb's Journal* bound volumes 5 through 7.

Volume 5 — 1980

January 41
February 42
March 43
April 44
May 45
June/July 46
August 47
September 48
October 49
November/December 50

Volume 6 — 1981

January 51
February 52
March 53
April 54
May 55
June 56
July 57
August 58
September 59
October 60
November 61
December 62

Volume 7 — 1982

January 63
February 64
March 65
April 66
May 67
June 68
July 69
August 70
September 71
October 72
November 73
December 74

1802
  cross assembler [CP/M] 5/432
  Elf 5/122
2716
  EPROM programmer [M-80] 6/189–190
6502
  assembler for the 6502 6/369–373
  byte-count routine 6/83
  improved timing 7/111–112
  opcodes for analysis [BASIC/PET] 6/96–97
  reader response 6/185
6800
  cross assembler [CP/M] 5/432
  video display program 5/439
6809
  multi-byte opcodes 6/279
  Tiny BASIC for the 6809, Santa Barbara 6/202–203
68000
  cross assembler 7/427–438, 7/488–498
  interfacing to an AIM 65 7/8–12, 7/29–31
  mnemonics 7/449–453
  multi-6800s 7/244–247
  multi-tasking kernel 7/1 5
8080 (*see also* Z80) 6/187
  /8086 cross assembler 7/156–164
  /Z80 indirect addressing 5/289; 5/394
  and CP/M 5/396
  and Z80 5/289; 5/394
  fast fourier transform 5/164
8080 *(contd.)*
  memory chip tester 6/533–534
  North Star DOSMOVER 6/124
  Pidgin 7/116
  PILOT 5/82
  registers in 47 bytes 5/173
  Small-C compiler 5/176
  translator system (to Z80) 5/216
  unaltered 5/42
  variations 5/173
  Z80
    cross assembler for 8080/Z80 6/230–234
    editor, assembler, disassembler in XL package 6/36–40
8086 (*see also* 8080)
  FORTH assembler 7/61
  trig 7/462–464
8087 7/361–363
8813
  polymorphic math functions 5/125
  9 or 16 bits? [Opcode design] 5/373
9900 5/46; 5/108

Acoustic coupler 5/351
Adams, E. 6/514
Addendum
  Newbury, S. 5/395
Addition trick 5/125

Addressing, indirect
6502 5/333
8080/Z80 5/289; 5/394
Adler, R. 5/443
ADM-3A 5/56
Adven-80, adventure development system 6/468–473
AIM 65/68000 interface 7/8–12, 7/29–31
Airy, B. 6/6, 6/7, 6/11
Algorithm to convert integer date [MP/M] 6/201
Algorithms 5/130; 5/169; 5/207; 5/264; 5/350
FORTH 6/148–149
numerical 7/309
vector generation 7/550–551
Allison, D. 5/73; 5/169; 5/207; 5/264; 5/350
Alpha listing H-8 5/104
ALS-8 to CP/M conversion 5/24; 5/261
Animal 5/350
Apple
user interface 5/267
Apple II
file system, Tiny 6/240–241
program renumbering 5/67
ROM byte-finder 5/243
Argum—A C command line processor 7/331–347
Arrays
cassette I/O routine [BASIC] 6/150–152
FORTH 5/242
North Star 5/40
Artificial intelligence 6/419
"Thinking" program, ERGO 6/100
Artificial intelligence language, LISP 6/526–530
Assemblers
6800 + 1802 cross assembler [CP/M] 5/432
assembler for the 6502 [FORTH] 6/369–373
cross assembler for the 8080/Z80 6/230-234
editor, assembler, disassembler in XL package 6/36–40
EDITOR/ASSEMBLER PLUS 6/123
FORTH 8086 7/61–33, 7/72–85
table driven 5/59; 5/171
Z80, subroutines 5/159

Bach, S. 5/243; 5/267
Balcom, O. 5/39
Barker, L. 5/42; 5/173; 5/205; 5/222; 5/268; 5/309
Barnhart, J. 7/388
BASIC 5/42
arrays, cassette I/O routine 6/150–152
BASIC-FORTH 6/427–428
branches 7/101–103
IBM PC BASIC 7/278
LLL BASIC 7/248
memory dump 5/260
memory management 7/500
MIN and MAX definition 6/312
North Star
cross-reference 5/147; 5/151; 5/339; 5/382
debugging concatenated programs 6/64
DOSMOVER 6/124
patches 5/40; 5/171
small VM development, portable software 6/492–495
statement generator 5/418
string arrays 5/40
terminal programs for 6/487
text editor input routine 6/42–43
time keeping in microcomputers 6/287–289
two column display 5/259
PET
analysis 6502 opcodes 6/96–97
speed reading using a computer
POKE and PEEK, BASIC version 3.2 enhancement 6/62
BASIC *(contd.)*
Poly 88 5/141
polynomial evaluation 6/146–147
Rubik's cube simulator 6/291–292
structured 5/201
time delay circuit design 6/197
Tiny BASIC 5/201; 5/356
Tiny BASIC for the 6809, Santa Barbara 6/202–203
bugs 6/419
ZX80 Sinclair 6/60–61
BASIC-FORTH [FORTH] 6/427–428
BASIC-Plus 6/45
Bassett, S. 6/148
Bender, A. 5/209
Bender, A.L. 7/116
Bentley, J. 5/130
Bertollo, D. 5/348
Binary to decimal 5/73
conversion routine 6/271
reader's improvements 6/422
Blum, R. 7/302
Blessing, R. 6/487
Bokelman, R. 5/126
Bolton, B. 6/330
Bomberger, A. 7/86, 7/175, 7/546
Borochoff, R. 5/126
Brodgen, W. 6/237
Bronstein, M. 5/122
Brown, A. 5/443
Brown, D. 5/67
Bugs
Small-C 5/83
SAM 76's CP/M 5/83
Bulletin boards 7/224–225
Byte
ROM byte-finder [Apple II] 5/248
Byte-count routine [6502] 6/83
Byte manipulation and pointers [Fortran] 6/103

C language (*see also* Small-C compiler) 5/15; 5/176; 5/191
command line processor 7/331–347
cross reference generator 7/266–270
time functions 6/330–331
C, Small (*see* Small-C)
C, Tiny (*see* Tiny-C)
Caeser, A. 5/309
Cage, T. 6/124
Cain, R. 5/176; 5/312
Calculators 5/392
Calendar routines 6/313, 6/516
Card shuffling 6/356
Carter, J. 5/159; 5/267
Cason, R. 5/127
Cassette interface 5/13
Caulkins, D. 5/230; 5/284; 6/63, 6/95; 7/145
CBASIC-2
DB.BAS package file maintenance 6/80
CDOS 5/25/394
Chamkis, J. 7/188, 223
Check digits 7/171–172
Checksums 7/302–303
CLOAD magazine, Inc. 6/122
Colley, W. 5/432
COMAL-80 [CP-M] 6/235–236
as an interpreter 6/235
Compal-80 6/102
Compiler writing tool, META 4 language 6/321
implementing an M4 interpreter 6/329
M4 language 6/325
META 4 language 6/322

Compilers
META II 5/169
Runic: An interactive, extensible compiler [FORTH] 7/106–108, 7/184–186
Small-C 5/176; 5/312; 7/520–546
error run-time library routine 6/53
bug fixes and enhancement 6/228
cross-compiler implementation 6/66–67
programming 6/504–507
Tincmp, compiler for Pidgin 6/284–286
Computer conferencing 6/429–431
Conley, W. 5/352
Connell, J. 7/486
Conversion
binary to decimal conversion routine 6/271
file conversion, H-8 text editor to CP/M 6/239
submit file comments 6/200, 6/415
Conversion routines
ASCBIN and BINASC [8086 assembly language] 7/413–414
Conversions/converters
ALS-8 to CP/M 5/24
Binary to decimal 5/73
CP/M to UCSD Pascal 5/367
Core, G. 5/443
Coroutines 7/3–5, 7/26
Corrections
Barker, L. 5/309
Keay's table 5/41
Parsons, R. 5/83
Silverston, S. 5/172
Cortesi, D. 6/200, 6/271, 6/312, 6/414, 6/422, 6/520; 7/6, 7/101, 7/152, 7/197, 7/237, 7/278, 7/280, 7/329, 7/373, 7/424, 7/471, 7/503
COSMAC 5/282
CP/M 5/11; 7/277, 7/317
6800 + 1802 cross assembler 5/432
ALS-8 to CP/M file converter 5/24; 5/261
bug [SAM 76's CP/M] 5/83
and CDOS 5/29; 5/394
COMAL-80 6/235-236
DDT, circumventing 6/200
downloading modem with MBOOT3 7/454-461
DUMP utility readability enhancement 6/93
file conversion, H-8 text editor to CP/M 6/239
game 5/406
Hardcopy device driver program 5/341
interrupts 7/546-549
keyboard input definition 6/194–196
linking to Morrow's cassette interface 5/13
NEW USER 7/217–222
OSI's CP/M 5/205
PEN 5/34
print utility 7/256–265
QUITIF 7/373–376
SCRIBE IX, word processing package 6/220-222
software 5/4; 5/32
submit file comments 6/200
table-driven assembler 5/59
text editor WORD-MASTER with the Compal-80 6/102
tiny interpreter [CP/M flavored C] 5/15
to UCSD Pascal file conversion 5/367
Z80, 8080, and CP/M 5/396
ZDM debugger 6/186
CP/M-86
vs. MSDOS 7/280–291
vs. PC-DOS 7/501
Crimmins, J. 7/314
Crockett, T. 5/79
Crockford, D. 7/93
Cross assembler for 8080/Z80 [8080/Z80] 6/230–234
XASM-86 features 6/232
Z-88 project 6/230
Cross assemblers 5/432
68000 7/427–438, 7/488–498
8080/8086 7/156–164
Cross reference generator [C] 7/266–270
Culleton, J. 5/125; 5/352

DaCosta, F. 5/39
Daetwyler, D. 7/256
Darling, M. 5/445
Database design [FORTH] 6/398–401
Davis, D. 6/126
DB.DBAS package file maintenance [CBASIC-2] 6/80
DDT, circumventing [CP/M] 6/200
Dea, R. 5/216
Deane, R. 5/242; 5/434
Debuggers
DDT, circumventing 6/200
ZDM debugger for CP/M users 6/80-81
Debugging concatenated programs [BASIC/North Star] 6/64
Decompiler [FORTH] 6/403
Demas, J. 6/36
Design
algorithm 5/130
opcode 5/373
Diablo 5/268
Digital systems DDOS 5/127
Displays
6800 5/439
North Star 5/259
Dobyns, B. 6/178
Documentation 5/426
Double density system 5/339
Douds, C. 5/127
Drexler, K. 5/439
DUMP utility readability enhancement [CP/M] 6/93
implementation 6/227
Dump/Dumping
BASIC 5/260
Duncan, R. 6/69, 6/140, 6/187, 6/403; 7/61, 7/292, 7/361, 7/413, 7/417, 7/501, 7/515
Dye, J. 5/309; 5/444

Ecklein, D. 5/394
Editors (*see also* Screen editors, Text editors)
editor, assembler, disassembler in XL package [8080/Z80] 6/36-40
EDITOR/ASSEMBLER PLUS [TRS-80] 6/123
program editors, reassessment of 237–238
screen editor, MINCE 6/178
screen-oriented editors 6/383
text editor input routine [BASIC] 6/42–43
text editor WORD-MASTER with the Compal-80 6/102
underline modification 6/126
ZX80 Sinclair 6/60–61
Electronic phone book [PCNET project] 6/63, 6/87, 6/133, 6/176
Elf 5/268
Elizondo, F. 5/261
Embedding wide characters within an ordinary line 6/515
Emery, G. 7/59
Enhancements
ADM-3A 5/56
entry points 5/267
equations, differential 5/392
Errata of computer programming 5/244
Errors 5/309; 5/349; 5/446

Faulkner, D. 6/15
Favitta, M. 6/80
Fibonacci number sequence [FORTH] 6/9, 6/91
Fiddes, L. 5/394
Fifth generation computers 7/552
File comparator [Z80] 6/15
File conversion (*see* Conversion)
  H-8 text editor to CP/M [conversion] 6/239
File system, Tiny [Apple II] 6/240–241
File transfer 5/375
File transfer utilities 7/355–360
Flashwriter I 5/406
Floppy disk test [Z80] 6/140–145
Foreign language trainer 5/423
Formatter, TI 9900 5/46; 5/108
FORTH 5/242; 5/434
  8086 assembler 7/61–63, 7/72–85
  algorithms 6/148–149
  assembler for the 6502 6/369–373
  BASIC-FORTH 6/427–428
  database design 6/398–401
  decompiler 6/403
  description 6/411–413
  fibonacci numbers 6/9
  fibonacci numbers sequence 6/91
  FORTH as language and environment 6/381–382
  formal definition 7/64–71
  floating-point package 7/377–385
  overview 6/6, 6/12
    construction of program-like commands 6/8
    metamorphic changes by concatenation of characters 6/10
    primitives 6/7
    reader response 6/137, 6/185
    use by novices 6/12
    variables stack 6/7
    parameterized command concept 6/54–59
      reader response 6/186
  philosophy of 6/363–368
    software development methodology 6/364
    software tools 6/367
  programming style 7/386–387
  recursion 7/59–60
  relocating loader 7/388–292
  screen editor for H-19 7/393–397
  screen-oriented editor 6/383
  string routines 6/51
  Z8000 FORTH 7/398–412
FORTH as a language and environment [FORTH] 6/381–383
FORTH description 6/411–413
Fortran
  byte manipulation and pointers 4/103
  Small-C cross-compiler implementation 6/66–67
Frampton, H. 5/24
Franz, M. 7/106, 7/184
Freed, E. 6/103
Funkhouser, R.K. 7/313
Fusfield, R. 5/168
Fusina, L. 7/8

Gabrielson, M. 5/443
Gale, W. 6/240, 6/282, 6/321
Gale, W.A. 7/476
Games, CP/M 5/406
Gammell, J. 5/40
Garcia, L. 5/418
Gaugler, G. 5/345; 5/423
Gelles, A. 5/85
Gibson, T. 5/201
Gintis, H. 7/439
Glenn, G. 7/248
Glitch 5/222
Go: utility command under CP/M 5/11
Gordon, H. 5/324; 5/373; 5/395; 6/6, 6/54
Gordon, H.T. 7/309
Gramer, J. 5/125
Granuzzo, C. 7/8
Graphics
  Flashwriter I [CP/M] 5/406
  microprocessor-assisted system 5/79
Graphics, IBM PC 7/292-301
Greenlaw, D. 6/88
Grigonis, R. 7/552
Guthery, S. 5/201

H-8 computer 6/239
Hacking 6/510–514
Halfant, M. 7/316
Hallem, R. 6/149
Hallen, R. 5/34
Hamlin, C.L. 7/241, 7/304
Hamlin, G. 5/70
Hardcopy device driver programs [CP/M] 5/341
Hardware reviews
  Hayes Stack Smartmodem 7/231
  Sinclair ZX81 7/188
  Universal Data Systems 103LP 7/145
Hardware simulator 7/241–243, 7/304–308
Harris, D. 6/63, 6/177
Harris, K. 6/363
Hashing 7/471
Hatfield, J. 6/122
Haydon, G. 6/398
Head, G. 5/222; 6/80; 6/220; 7/95, 7/143, 7/217, 7/454
Heath
  H-8 5/104; 5/287; 7/109-110, 7/122–123, 7/152–153
  H-14 5/201; 5/332; 5/395
  H-19 7/329
Hemminger, C. 5/309
Henderson-Gragg, R. 7/231
Hendrix, J. 6/492
Hendrix, J.E. 7/20, 7/520
Hex keyboard 5/122
Hightower, M. 6/127
Hinds, J. 6/202
Hoffman, R. 146
Holtrichter, J. 5/339
Horizon [Interfacing H-14] 5/395
Howell, J. 5/173; 5/266; 5/396
Hudson, G. 5/259
Hudson, L. 5/40; 5/382
Hudson, R. 5/356
Hughes, L. 5/375; 7/355
Hypotenuse approximation on integer values 6/200
  reader input 6/312, 6/359, 6/520

IBM PC 7/237
  BASIC 7/278
  custom character sets 7/515–519
  graphics 7/292–301
Improvements to:
  Apple user interface 5/276
  CP/M and CDOS 5/394
  Zant's modular programming 5/266
Indirect addressing [8080 and Z80 5/289] 5/333
Ingerman, P.Z. 7/172

Installation, Phantom-deselect 5/42
Interface
  Apple 5/67
  CP/M linked to Morrow's cassette interface 5/258
  H-14 to Horizon 5/395
  Wh 14 printer to serial POR 5/443
Interpreter
  LISP 5/268
  Tiny 5/15

James, J. 6/429
James, J.S. 7/202
Jamieson, C. 5/83
Johnson, D. 5/84
Johnson, J. 6/514
Johnson, R. 5/171
Johnson, S. 5/197

Kalish, M. 6/122
Kan, M. 7/211, 7/348
Kanefsky, R. 6/514
Keay, C. 5/41
Kernigham, B. 5/191
Keyboard, Hex 5/122
Keyboard input definition [CP/M] 6/194–196
  theory 6/195
Keyplus integrated utility 5/309
Kildall, G. 5/2
Knippenberg, H. 5/260; 6/197
Knuth, D. 5/88; 5/244
Konopik, N. 5/234; 5/332
Kossow, A. 7/244, 7/427
Kruse, R. 5/272; 5/395
Kudlac, S. 6/514

Language register 6/3
Languages (*see also* individual names)
  C 5/191
  foreign 5/423
  META II 5/142
  syntax-oriented compiler writing language 5/142
Las Vegas super slot 5/406
Lawrence, J. 6/45
Laxen, H. 6/383
Lesk, M. 5/191
LET processor [LLL BASIC] 7/248–255
Lindenschmitt, G. 5/84
LISP (*see* artificial language)
LISP, interpreter 5/268
Listing, H-8 5/104
Loeliger, R. 6/381
Logic 5/348
Long, W. 5/371
Loughman, W. 5/431
Love program 5/345
Lucky, R. 5/171; 5/309
Lunsford, D. 5/172

M4 interpreter 6/329
M4 language 6/325
M-80
  2716 EPROM programmer 6/189–190
Maclaughlin, J. 5/446
Maggs, P. 6/127
Mailing list label printer [SLIC] 6/149
Martin, T. 5/443

Math functions [8813] 5/125
Mathematical typography 5/88
Maurer, W. 5/333; 5/445
Maurer, W.D. 7/111, 7/449
MCALL-C 5/395
McGaw, J. 5/125
Measurement standards 7/93–94
Memory chip tester [8080] 6/533–534
Memory dump [BASIC] 5/260
Memory test [Z80] 6/69
Memory-mapped video displays 6/16
META 4 (*see* Compiler writing tool)
Michalski, M.W. 7/550
Microchess 5/126
Microprocessors (*see* individual names)
Microsoft 5/222
Miller, E. 6/189
Miller, W. 5/13
Mills, J. 6/230; 7/156
MIN and MAX definition [BASIC] 6/312
  extended to other functions 6/354
MINCE, screen editor 6/178
MITS DOS disk to disk copy routine 5/209
Mnemonics [68000] 7/449–453
Modems
  downloading with MBOOT3 [CP/M] 7/454–461
  program for [FORTH] 7/202–207
Modular programming 5/266
Monitor MON 3 [TRS-80] 6/122
Monitors
  Elf tiny 5/268
  SYM-1 5/324
Monroe, A.J. 7/377
Moore, F. 5/426
Morrow, G. 5/13
Moss, W. 5/104
MOVIT 5/431
MP/M
  algorithm to convert integer date 6/201
MSDOS vs. CP/M-86 7/280–291
Multiple file transfer 5/375
Munro, J. 5/443

N-LOGS 5/400; 6/53, 6/281
Naess, O. 5/364
Networks 5/226; 5/230; 5/257
Newberry, S. 5/42; 5/125
NIBASM 5/371
Nichol, R. 6/123, 6/526
Nicholson, L.K. 7/91
North Star (*see* BASIC; Horizon)
  DOS 6/36, 6/270
  DOSMOVER [8080] 6/124
  terminal programs for 6/487
North Star Horizon
  connection to Epson printer 7/208–210
  timer routine 7/86–90
North Star VOS 7/20–23, 7/25–28
Number language [N-LOGS] 5/400

Oliver, J. 5/39
Odette, L. 7/398
Opcode design 5/373
Operating systems (*see* individual names)
OSI 5/205
Output [COSMAC 1861] 5/282
Ouverson, M. 7/224

Palmer, J. 6/60, 6/194
PAN protocol revision [PCNET project] 6/63
Parameterized command concept [FORTH] 6/54–59
  access 6/55
  conditional-jump primitive 6/57
  data structures 6/55
  notation 6/54–55
  reader response [FORTH] 6/186
  status-register 6/56
Pascal 5/75; 5/127; 5/367
  card shuffling 6/356
  "upside" down, nature of 6/4
Patches
  POKE and PEEK, BASIC version 3.2 enhancement 6/62
Patches, North Star DOS 5/40
Pattern classifications 7/114–115, 7/124–126
PC-DOS vs. CP/M-86 7/501
PCNET 5/230; 5/284
PCNET project 6/172
  electric phone book 6/63, 6/87, 6/133, 6/176
  PAN protocol revision 6/63
  PCNET protocol 6/63, 6/177
PDP-1802 hardware simulator 7/241–243, 7/304–308
Pehoushek, J. 6/291
Pelle, M. 5/235
PEN 5/34
Performance evaluation 7/348–354
Personal Computer Networks (*see* PCNET)
PET 5/127
Peter, K. 6/514
Pfafman, F. 5/42
Phelan, C 5/309
Phillips, T. 5/443
Philosophy of FORTH [FORTH] 6/363–368
PHIMON subroutines 6/15
Phone dialer 5/84
Pidgin 6/321
  for 8080 [CP/M] 7/116–121, 7/127–140
  tests 7/476–485
  tincmp, compiler for Pidgin 6/284–286
  Z80 macro-assembly implementation 7/439–449
Pidgin, systems programming languages 6/282–284
PILOT 5/82
Pocket computer, Radio Shack 6/88
POKE and PEEK, BASIC version 3.2 enhancement [BASIC] 6/62
  alternative patch 6/228
Poly 88 BASIC 5/141
Polymorphic 8813 math functions 5/125
Polynomial evaluation [BASIC] 6/146–147
Portable software development, Small VM 6/492–495
Price, G.H. 5/282
Print utility [CP/M] 7/256–265
Printer [H-14] 5/332
Printers
  printer toggle installation in North Star DOS 5.2 6/270
Probes, logic 5/348
Program editors, reassessment of [editors] 6/237–238
Programmers
  2716 EPROM programmer 6/189–190
Programming
  Artificial Intelligence language, LISP 6/526–530
  BASIC 5/341
  BASIC-FORTH 6/427–428
  C and Tiny C 5/201
  C language 5/191
  CP/M 5/341
  changing SPI to CP/M 5/32
  errata 5/244
  pastimes and pleasures 5/37; 5/285; 5/330
Programming *(contd.)*
  Pidgin, systems programming language 6/282–284
  Small-C programming 6/504–507
  time functions 6/330–331
  Zant's modular programming 5/266
Programs
  6800 5/439
  68000 cross assembler 7/429–438, 7/488–498
  68000/AIM interface 7/29–31
  8080/8086 cross assembler 7/156–164
  8086 trig 7/463–464
  8086 assembler [FORTH] 7/61–63, 7/72–85
  adventure development system, Adven-80 6/468–473
  assembler for the 6502 6/369–373
  BASIC-FORTH 6/427–428
  C command line processor 7/331–347
  COMAL-80 6/235–236
  compiler writing tool, META 4 6/321
  conversion routines [ASCBIN and BINASC] 7/414
  CP/M 5/406
  CP/M + CDOS compatability 5/29
  cross assembler for 8080/Z80 6/230–234
  cross-reference generator [C] 7/266–270
  custom character sets [IBM PC] 7/518–519
  debugging concatenated programs 6/64
  decompiler 6/403
  downloading modem [CP/M] 7/454–461
  DUMP utility readability enhancement 6/93
  file comparator 6/15
  file conversion, H-8 text editor to CP/M 6/239
  file system, Tiny 6/240–241
  file transfer utilities 7/355–360
  floppy disk test 6/140–145
  FORTH floating-point 7/381–385
  FORTH screen editor 7/395–397
  FORTH style examples 7/387
  H-8 front panel 7/122–123, 7/113
  Hash 'n Pascal 7/508–510
  keyboard input definition 6/194–196
  LET processor [LLL BASIC] 7/248–255
  love program 5/345
  mailing list label printer 6/149
  memory test [Z80] 6/69
  modem program [FORTH] 7/202–207
  multi-tasking kernel [68000] 7/5
  NEW USER [CP/M] 7/217–222
  Newton's square root 7/502
  North Star DOSMOVER 6/124
  North Star-to-Epson connection 7/208–210
  pattern classification 7/124–126
  performance evaluation 7/348–354
  Pidgin 7/441–449, 7/477–485
  Pidgin, systems programming language 6/282–284
  portable screen-oriented editor 7/32–51
  print utility [CP/M] 7/256–265
  QUITIF [CP/M] 7/373–376
  relocating loader [FORTH] 7/390–392
  Rubik's Cube simulator 6/291–292
  screen-oriented editor 6/383
  Small-C compiler 7/520–546
  Small-C for the 9900 7/316–322
  Small-C Diskdoc 7/165–171
  Small-C programming 6/504–507
  Small Shell 7/20–23, 7/25–28
  speed reading using a computer 6/95
  spelling checker 7/177–183
  system clock 7/211–216
  terminal programs for North Star 6/487
  timer routine [Horizon] 7/88–90
  user interface to Apple II 5/267

Programs *(contd.)*
video graphics [IBM PC] 7/292–301
wildcards [UNIX] 7/473–475
Z8000 FORTH 7/401–412
Protocol 5/234
Pugh, T. 5/234
Purdin, T. 5/151

Quick-key 5/418
Quinn, C. 5/39

Radio Shack (*see* TRS-80)
Radio Shack pocket computer 6/88
Raeside, D. 5/121
Raeside, D.E. 7/114
Ragsdale, W. 6/369
RAMcard with RAMdrive 7/462
Ream, E.K. 7/13
Reentrant formatter [TI9900] 5/108
Register/storage [8080] 5/173
Rennard, R. 5/444
Ritchie, D. 5/191
Rohde, A. 6/150
ROM byte-finder [Apple II] 5/243
Roth, R. 6/504
Routines
alternate 5/445
arrays, cassette I/O routine 6/150–152
binary to decimal conversion routine 6/271
byte manipulation and pointers 6/103
calendar routines 6/313
database design 6/398–401
DUMP utility readability enhancement 6/93
MITS DOS disk to disk copy 5/209
polynomial evaluation 6/146–147
Small-C cross-compiler implementation 6/66–67
text editor input routine 6/42–43
text editor WORD-MASTER with the Compal-80 6/102
Rowland, D. 5/269
Rubik's Cube simulator [BASIC] 6/291–292
Rubin, S. 5/349; 6/100
Rules 5/309
Runic: An interactive, extensible compiler [FORTH] 7/106–108, 7/184–186
Runtime library [Small-C] 5/312

S.D. board 5/257
SAM 76 5/83
Santa Barbara (*see* Tiny BASIC for the 6809)
Saunders, W. 5/395
Scargill, P. 6/468
Schorre, D. 5/142
Scientific computer 5/400
Scott, J. 5/443; 6/238
Screen editors 6/178
FORTH screen editor 7/393–397
MINCE 6/178
addendum 6/227
extended with Amethyst 6/181–182
portable screen-oriented editor 7/13–19, 7/27–28
Screen-oriented editor 6/102
[FORTH] 6/383
SCRIBE IX, word processing package [CP/M] 6/220–222
Serial port 5/425
Shoemaker, D.C. 7/109, 7/187
Shoer, L. 7/208
Silverston, S. 5/172
Simons, K. 5/400
Sinclair BASIC 6/61
Sinclair ZX81
building 7/486–487
direct video input 7/223
Sippy, L. 5/425
Sjowall, T. 5/367
Skjellum, A. 5/11; 5/289; 6/16, 6/102, 6/423; 7/331, 7/472
SLIC
mailing list label printer 6/149
Small-C
8080 5/176
for 9900 7/316-322
programming 6/504–507
repair/maintenance utility 7/165–171
runtime library 5/324
Small-C compiler 5/312; 7/520–546
bug fixes and enhancements 6/228
error run-time library routines [compiler] 6/53
modification 6/465
Small-C cross-compiler implementation [Fortran] 6/66–67
bug correction 6/67
bugs 6/228, 6/277
update 6/104
Small Shell 7/20–23, 7/25–28
Small systems and monitors 5/324
Small VM development, portable software [BASIC/North Star] 6/492–495
Smart, B. 6/270
Smith, B. 5/267
Smith, D. 5/443
Smith, G. 6/42, 6/64
Smith, W. 5/240
Softky, S. 6/100
Software
CP/M 5/4; 5/445
software market 5/309
Software reviews
CLOAD Magazine, Inc. 6/122
COMMWHIZ 7/226
DB.BAS package file maintenance 6/80
editor, assembler, disassembler in XL package 6/36–40
EDITOR/ASSEMBLER PLUS 6/123
JRT Pascal 7/503
monitor MON 3 6/122
PIE 7/187
screen editor, MINCE 6/178
SCRIBE IX, word processing package 6/220–222
the WORD 7/143
the Working FORTH Learning System 7/417
utility package, Disk Keyplus 6/122
Volkswriter 7/313
Write-On 7/314
ZDM debugger for CP/M users 6/80–81
ZEN 7/95
Solntseff, N. 7/386
Sonderstrup, S. 6/235
Source-text area, extended 5/371
Speech synthesizer interface [TRS-80] 6/127–129
Speed reading using a computer [BASIC/PET] 6/95
using ED and PIP in CP/M 6/186
Spelling checker [CP/M] 7/175–183
SPI formatted programs 5/32
Sprenkle, J. 5/279
SR-52 programmable calculator 5/392
Statement generator [North Star] 5/418
Stewart, H. 5/445
STOIC [North Star disk] 5/55
Stowe, T. 5/287
String arrays [North Star] 5/40

String routines [FORTH] 6/51
Submit file comments [CP/M conversions] 6/200, 6/415
Subroutines [Z80 assembler] 5/159
SYM-1 5/324
Symbol table alpha listing [H-8] 5/104
Syntax-oriented compiler writing language 5/142
System Clock 7/211–216

Table driven assember 5/59; 5/171
Taylor, J.L. 7/266
Tektronix 4010 simulator [Z80] 6/423-425
Terminal parameters [Pascal] 5/75
Terminal programs for North Star [North Star] 6/487
Testing binary integer for single 1-bit 6/414
Text editor
  input routine [BASIC/North Star] 6/42–43
  WORD-MASTER with the Compal-80 [CP/M] 6/102
Text editors 7/91–92
Thagard, N.E. 7/248
"Thinking" program, ERGO [artificial intelligence] 6/100
  documentation 6/319
Thompson, B. 5/223
TI 9900 5/45; 5/108
  TI PMX 5/46; 5/108
Time functions (C language) 6/330–331
Time-delay circuit design [BASIC] 6/197
  small error 6/420
Time-keeping in microcomputers [BASIC/North Star]
    6/287–289
Timin, M. 6/411
Tincmp, compiler for Pidgin 6/284–286
Ting, C.H. 6/427; 7/64
Tiny BASIC (*see* BASIC)
  for the 6809, Santa Barbara [6809] 6/202–203
Tiny interpreter 5/15
Tiny monitor [Elf] 5/268
Top-down 5/352; 5/443
Translator [8080 to Z80] 5/216
TRS-80 5/441
  EDITOR/ASSEMBLER PLUS 6/123
  Monitor MON 3 6/122
  synthesizer interface 6/127–129
  utility package, Disk Keyplus 6/122
Turing machine simulation [programs] 6/45
Typography, mathematical 5/88

UCSD Pascal (*see* Pascal)
Underline modification 6/125
Universal language 6/45
UNIX, Expanded wildcards 7/472–475
Utilities
  command under CP/M 5/11
  file transfer 7/355–360
  multiple file transfer 5/375
  NEW USER [CP/M] 7/217-22
  print [CP/M] 7/256–265
  Small-C repair/maintenance 7/165–171
Utility package, Disk Keyplus [TRS-80] 6/122

Van Buer, D. 5/59; 5/171; 5/268
Van Iderstine, T. 6/62
Vector generation algorithm 7/550
VDM-1 6/124
Video display RAM 5/439
Video drivers [Z80] 6/16–17
  time-dependent coding 6/93
Vocabulary trainer 5/423

Wakerley, J. 5/56
Walters, N. 5/46; 5/108
Warner, J. 5/29
Wasserman, P. 5/172
Wegner, K.F. 7/114
Wetherell, C. 5/37; 5/77; 5/285; 5/330
Wexler, S. 5/309; 6/122
WH 14 printer 5/425
Whitlock, H. 6/96
Wilcox, R. 5/82
Wildcards [UNIX] 7/472–475
Willoughby, S. 5/341
Woodhull, A.S. 7/393
Woods, P. 6/66, 6/104
Word processing package (*see* SCRIBE IX)

X-Ref tool [North Star] 5/339

Yakush, S. 5/392
Yates, D. 6/287

Z80 (*see also* 8080)
  assembler 5/159
  circuit (H-8) 5/287
  file comparator 6/15
  floppy disk test 6/140–145
  indirect addressing 5/289; 5/394
  memory test 6/69
  power increase 5/172
  tektronix 4010 simulator 6/423
  translator system from 8080 5/216
  video drivers 6/16–17
Z80 macro-assembly implementation 7/439–449
Z80 microprocessor, use and misuse 6/187–188
Z8000 FORTH 7/398–412
Zant, R. 5/266
ZDM debugger for CP/M users [Z80] 6/80–81
  inline assembler [CP/M] 6/186
Zeiger, M. 5/32
ZSID bug and patch 6/519
ZX 65 5/272; 5/395
ZX80, Sinclair [BASIC] 6/60–61
  reader response 6/137, 6/229

5494-7

THE DR. DOBB'S SERIES

The People's Computer Company

"There's more creativity, wild ideas, and raw enthusiasm here per page than you are likely to find anywhere else in the personal computing press." *Creative Computing*

DR. DOBB'S JOURNAL, Volume 1

Chronicles the advent of the microcomputer era in 1976. Always pertinent for bit crunching and byte saving, home-brew computer projects, and the technical history of home computing. Topics include: Tiny BASIC, the first words on CP/M, Speech Synthesis, Floating Point Routines, 6502 Disassembler for the Apple, and much more. #5475, paper, 368 pages.

DR. DOBB'S JOURNAL, Volume 2

The small computer emerges as a powerful tool for the modern age in these issues from 1977. Topics include: Lawrence Livermore Lab's BASIC, Dr. Starkweather's PILOT, Using a Modem, String Handling Techniques, Ciphers, Turtle Graphics, micro utilities, and much more. #5484, paper, 480 pages.

DR. DOBB'S JOURNAL, Volume 3

Explores the foundations of the mass market computer industry in 1978. Topics include: Programming in BASIC, PILOT, and Pascal, RAM Memory Testers, Apple utility programs, S-100 Bus Standard, STRUBAL: A Structured BASIC Compiler, and much more. #5490, paper, 480 pages.

DR. DOBB'S JOURNAL, Volume 4

Heralds the widespread acceptance of the microcomputer in America. Innovative ideas and articles guide the reader through the age of discovery in 1979. Topics include: Selecting Business Software, Microcomputer Speech and Music, Information Networks, interfacing techniques, and much more. #5491, paper, 480 pages.

DR. DOBB'S JOURNAL, Volume 5

Focuses on the technological promise of the modern microcomputer and the creative challenge facing programmers in 1980. Topics include: The revolutionary impact of CP/M, C programming and the UNIX operating system, survey of computer networks, software portability, introduction to FORTH, compiler writing, and much more. #5492, paper, 464 pages.

DR. DOBB'S JOURNAL, Volume 6

The microcomputer enters the mainstream–these issues assess the revolutionary impact of microcomputers on the individual and on our society in 1981. Topics include: Computer Conferencing, The Power of FORTH, 8- and 16-Bit Technology, Rubik's Cube Simulator, An Adventure Game Development System, and much more. #5493, paper, 576 pages.

DR. DOBB'S JOURNAL, Volume 7

Examines the potential of powerful 16-bit micros including the birth of the IBM PC in 1982. Topics include: In-depth coverage of FORTH, 68000, 8088 programming, C software tools, utilities for CP/M including a spelling checker, using bulletin boards, and more. #5494, paper, 592 pages.

HAYDEN BOOK COMPANY
a division of Hayden Publishing Company, Inc.
Hasbrouck Heights, New Jersey

ISBN 0-8104-5494-7